Interactive Visualization and Collision Detection using Dynamic Simplification and Cache-Coherent Layouts

by
Sung-Eui Yoon

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2006

Approved by:

Dinesh Manocha, Advisor

Ming C. Lin, Reader

Peter Lindstrom, Reader

Anselmo Lastra, Committee Member

Valerio Pascucci, Committee Member

ABSTRACT

SUNG-EUI YOON: Interactive Visualization and Collision Detection using Dynamic Simplification and Cache-Coherent Layouts (Under the direction of Dinesh Manocha)

Recent advances in model acquisition, computer-aided design, and simulation technologies have resulted in massive databases of complex geometric models consisting of more than tens or hundreds of millions of triangles. In spite of the rapid progress in the performance of CPUs and graphics processing units (GPUs), it may not be possible to visualize or perform collision detection between massive models at interactive rates on commodity hardware. In this thesis, we present dynamic simplification and cache-coherent layout algorithms for interactive visualization and collision detection between large models, in order to bridge the gap between the performance of commodity hardware and high model complexity.

Firstly, we present a novel dynamic simplification algorithm that efficiently handles massive models for view-dependent rendering while alleviating discontinuity problems such as visual poppings that arise when switching between different levels of detail (LODs). We describe an out-of-core construction algorithm for hierarchical simplification of massive models that cannot fit into main memory. We also apply dynamic simplification to collision detection and introduce a new conservative distance metric to perform fast and conservative collision detection between massive models. Our approach is conservative in that it may overestimate the set of colliding primitives, but never misses any collisions.

Secondly, we present novel cache-oblivious layout algorithms for polygonal meshes and hierarchies to minimize the expected number of cache misses for a wide variety

of applications. Our layout algorithms only assume that runtime applications have random, but cache-coherent access patterns. However, we do not require any knowledge of cache parameters such as block and cache sizes. We demonstrate benefits of our layout algorithm on three different applications including view-dependent rendering, collision detection, and isosurface extraction.

We have implemented our algorithms on a desktop PC and are able to achieve significant improvements over previous approaches and obtain interactive performance (more than 10 frames per second) on view-dependent rendering and collision detection between massive and complex models.

To my wife, Dawoon Jung

ACKNOWLEDGMENTS

I would like to acknowledge the enormous amount of help given to me throughout the course of the Ph.D. program. Above all, I would like to thank my advisor, Dinesh Manocha. This work would not have been possible without his excellent guidance and tremendous support.

I also would like to thank the rest of my committee members, Anselmo Lastra, Ming C. Lin, Peter Lindstrom, and Valerio Pascucci. My early work for the dissertation was drawn from portions of course projects in the classes of Anselmo Lastra and Ming C. Lin. Also, Peter Lindstrom and Valerio Pascucci granted me the opportunities to work with them at Lawrence Livermore National Lab. as a summer intern student, which provided me invaluable experiences to extend my research interests and gave me two exciting chances to travel the USA between California and North Carolina.

Also, I am grateful to many members of UNC computer science faculty and staff. Their often unnoticed, but always helpful support has enabled me to progress smoothly.

I would also like to thank other colleagues and coauthors: Bill Baxter, Russ Gayle, Naga Govindaraju, Martin Isenburg, Ted Kim, Young Kim, Jayeon Jung, Brandon Lloyd, Miguel Otaduy, Stephane Redon, Brian Salomon, Avneesh Sud, Gokul Varadhan, and Kelly Ward. They helped me many ways to overcome research obstacles that I encountered throughout my pursue of Ph.D. In particular, I would like to thank Brian Salomon, who worked with me in different projects and helped me in many aspects.

Also, I would like to thank Elise London and Charlotte Powell for their technical editing assistance. I would like to thank Mary Wakeford, whose efforts greatly enriched my understanding of American culture and spoken English.

Also, I am very grateful to my parents, brother, sister, and parents-in-law for their support from Korea. Finally, I cannot thank enough to my wife, Dawoon Jung. Without her constant support and care, I could not have enjoyed the life of a Ph.D student. Thank you!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Recent advances in model acquisition, CAD, and simulation technologies have resulted in massive databases of complex geometric models. Large meshes composed of tens or hundreds of millions of triangles are frequently used to represent CAD environments, isosurfaces, scanned models, and terrains. They are also used to design large scale simulations in complex environments such as large man-made structures. An example of a double eagle tanker as a CAD model is shown in Fig. 1.1.

Interactive visualization and collision detection are frequently used in computer-aided design (CAD), virtual prototyping, walkthrough of architectural models, path planning, large scale simulation, and scientific visualization.

In spite of the rapid progress in the performance of graphics processing units (GPUs) and CPUs, it may not be possible to interactively render such complex datasets or perform interactive collision detection between massive models on current commodity hardware. Moreover, the performance utilization of GPUs and CPUs is drastically decreasing as model complexity is increasing and, thereby, the access time to underlying representations of the models is also increasing. The increased access time is mainly caused by several orders of magnitude difference of access time between different levels of memory hierarchies including L1 and L2 caches, main memory, and the disk.

There have been substantial prior research efforts to bridge the gap between the high

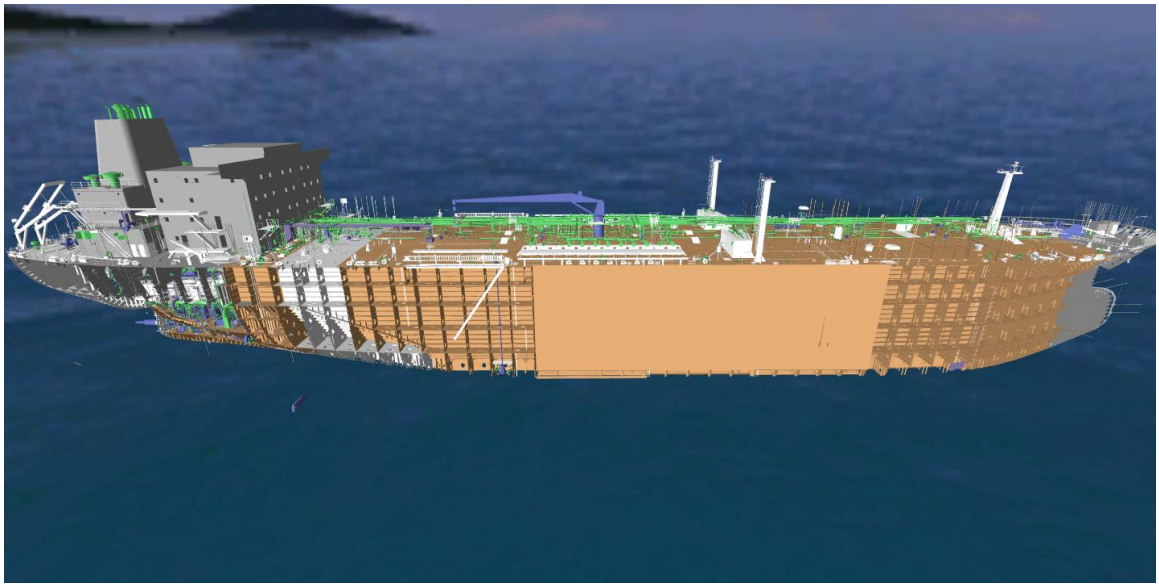


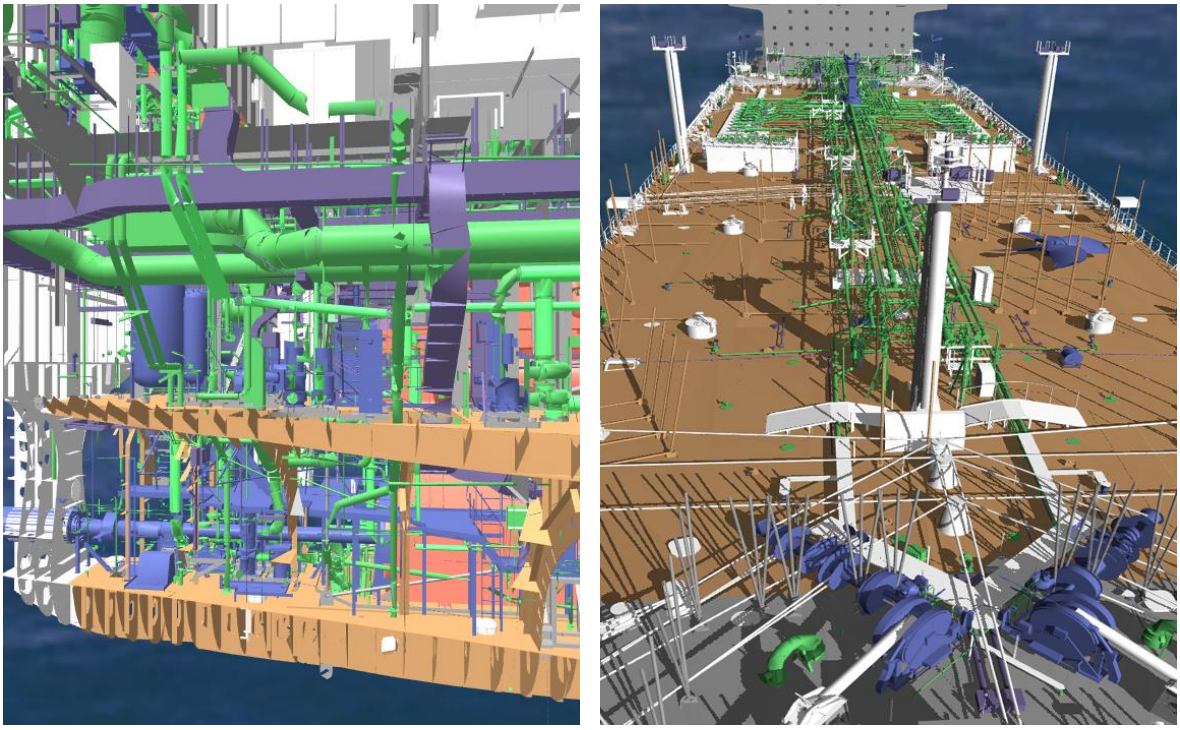
Figure 1.1: **An Example of Massive and Complex Model:** In this figure, a Double Eagle tanker consisting of 82 million triangles is shown. This model has a very irregular distribution of geometry; it has large area at the center with little geometry while other areas have high polygon density. This is more evident in the close-up of the engine room at the left bottom of the model shown in Fig. 1.2.

model complexity and performance of commodity hardware. These research efforts can be classified as the following:

- **Approximation algorithms:** The performance of applications can be improved by using approximate methods of original problems within an error bound instead of using exact computations. An example of an approximation algorithm is to use simplified geometry for rendering massive models.

- **Cache-coherent algorithms:** Cache-coherent access patterns of any algorithm are a critical component in reducing access time of the underlying data representation and, thereby, improving utilization of processing power of CPUs and GPUs.

- **Output-sensitive algorithms, etc:** If some portions of a mesh are not visible



(a) Engine room (b) Deck

Figure 1.2: **Close-ups of the Double Eagle Tanker: (a)** Engine room of the double eagle tanker, **(b)** Deck of the model with high quality shadows.

to the viewer, we can omit rendering these portions. This technique as an example of output-sensitive algorithms is called visibility culling. There are also parallel algorithms that interactively handle massive models by using multiple CPUs and GPUs.

In this thesis, we present novel dynamic simplification methods as approximate algorithms as well as cache-oblivious layout computations as cache-coherent algorithms for interactive visualization and collision detection between massive and complex models. Also, we propose a visibility culling algorithm as an output-sensitivity method for interactive visualization.

In this chapter, we will describe our two major applications, interactive visualization

and collision detection. Next, we will outline our research objectives and briefly consider prior work in dynamic simplification and cache-coherent layouts in order to analyze the challenges of dealing with massive and complex models. Finally, the thesis statement and organization of rest of the thesis will be presented.

1.1 View-Dependent Rendering

Interactive visualization of massive and complex models is a challenging problem in computer graphics and visualization. In order to achieve interactivity of visualization of massive models, view-dependent rendering has been widely used (Clark, 1976; Funkhouser & Squin, 1993; Hoppe, 1996). The main idea of view-dependent rendering is to use lower resolution on the portion of the mesh that is far away from the viewer. It is based on the assumption that the error introduced by the lower resolution of the mesh is imperceptible to the viewer. There are two different methods to implement the view-dependent rendering technique: static levels of detail and dynamic simplification.

- **Static levels of detail (LODs):** A few approximations of the mesh are precomputed at different levels of detail of the mesh. If the mesh moves far away from the viewer, a LOD with lower resolution of the mesh can be selected.

- **Dynamic simplification:** An LOD representation with the possible near-minimum number of triangles to meet an error bound specified by the user is computed at runtime. An acceleration data structure that encodes refinement and simplification of the mesh is used to efficiently compute the LOD representation at runtime.

The primary benefit of using static LODs is its very low computation overhead at runtime. However, switching between different LODs can result in noticeable visual popping artifacts at runtime. Dynamic simplification has been introduced to alleviate

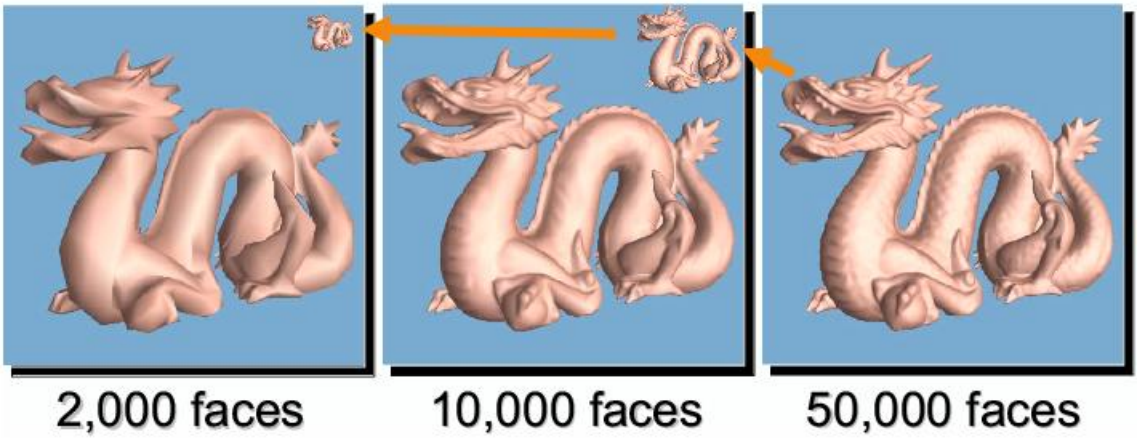
the visual popping artifacts because it provides a smooth transition between different LODs. On the other hand, the main disadvantage of dynamic simplification is its relatively high overhead to dynamically generate LODs to meet the error bound with the near-minimal number of triangles.

Hierarchical representations, such as vertex hierarchies (Hoppe, 1997), have been used to accelerate the performance of dynamic simplification; the hierarchy effectively represents a way to generate a simplified mesh of the original mesh at runtime. However, prior view-dependent computations based on these hierarchies have been reported as much slower than computations based on static LODs. This is mainly due to view-dependent computation to dynamically generate simplified geometry. An example of static LODs and dynamic simplification is shown in Fig. 1.3.

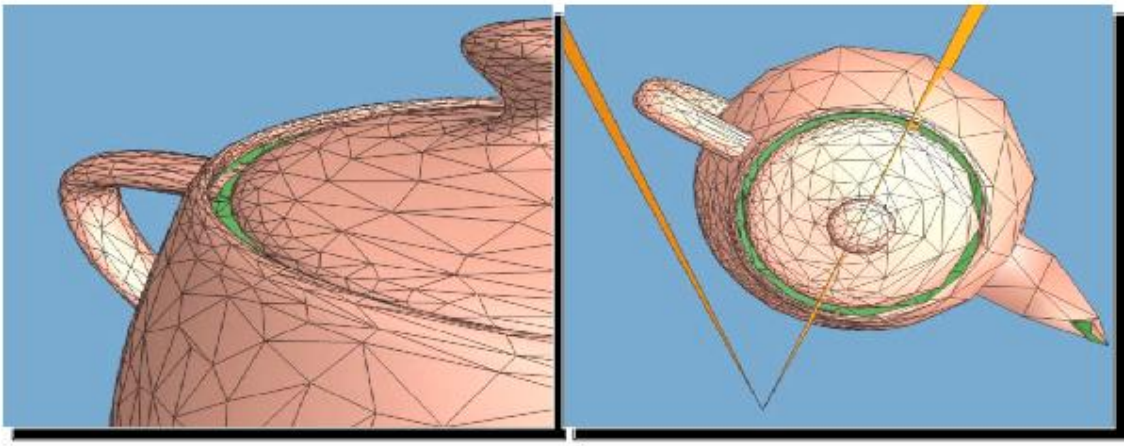
1.2 Collision Detection

Collision detection frequently arises in various applications including virtual prototyping, dynamic simulation, interaction, navigation and motion planning. Collision detection has been exhaustively researched for more than three decades (Jimenez et al., 2001; Lin & Manocha, 2003). Most of the commonly used algorithms are based on spatial partitioning or bounding volume hierarchies (BVHs). Some of commonly used BVHs include sphere-trees (Hubbard, 1993), AABB-trees (Beckmann et al., 1990), and OBB-trees (Gottschalk et al., 1996).

The performance of collision detection depends on the input model complexity and the problem output, which is the number of colliding or overlapping primitives. However, existing algorithms may not achieve interactive performance on large models consisting of tens or hundreds of millions of triangles due to their high complexity and output of the problem. Moreover, the memory requirements of these algorithms



(a) Static LODs



(b) Dynamic LODs

Figure 1.3: **Dynamic and Static LODs:** Three versions of static LODs of a dragon model are shown in figure (a). A dynamic LOD in the first person's view and the third person's view is shown at the left and right of figure (b) respectively. A low resolution LOD is used when the viewer is far away from the model. The viewer may notice severe popping artifacts when switching between different LODs. On the other hand, a dynamic LOD represents a mesh with a varying resolution over the mesh; popping artifacts can be reduced by providing a smooth transition between different LODs by using dynamic LODs. The images are courtesy of Hugues Hoppe's SIGGRAPH 97 talk slides.

are typically very high, as precomputed BVHs can occupy many gigabytes of space (Gottschalk et al., 1996). Moreover, the number of pairwise overlap tests between the bounding volumes can grow as a super-linear function of the model size, thereby slowing down the performance of collision detection.

Approximate Collision Detection: In order to achieve interactive performance of collision detection given high model complexity, approximate collision detection has been widely used. Hubbard (Hubbard, 1996) introduced time critical collision detection in order to provide interactive collision detection performance by using sphere trees. The sphere tree allows the time critical algorithm to progressively refine the accuracy of collision detection. It also provides interactive collision detection capability between polygonal meshes. Conceptually, all the bounding volume hierarchies such as AABB-trees (Beckmann et al., 1990), OBB-trees (Gottschalk et al., 1996), and k-DOP-trees (Klosowski et al., 1998) can support time critical collision detection. However, one of the main issues of performing approximate collision detection is how to quantify and reduce the collision detection error introduced by approximate collision detection. This has not been adequately addressed.

Recently, Otaduy and Lin (Otaduy & Lin, 2003) proposed *contact-dependent levels of detail (CLODs)*, which are precomputed dual hierarchies of static LODs used for approximate collision detection. The runtime overhead of this approach is relatively small because it utilizes static LODs. However, switching LODs between successive instances may result in a large discontinuity between outputs of collision detection caused by drastic changes of colliding triangles in the simulation. Moreover, the underlying approach assumes that the input model is a closed, manifold solid; therefore, it is not directly applicable to other general data sets including polygon soups.

1.3 Objectives

In this section we explain the objectives of our research. These include:

- **Interactive performance:** One of our key goals is to design algorithms providing interactive visualization and collision detection between massive and complex models. We consider that an algorithm provides interactive performance if its runtime performance is more than 10 frames per second.

- **Generality:** Since we want to handle a wide variety of polygonal meshes, we attempt to design algorithms that do not assume any particular geometric or topological structure of polygonal meshes. This goal is particularly important when dealing with CAD data sets since some CAD models have multiple degenerate triangles and even exist as polygon soups. As an example of a massive CAD model, a Double Eagle tanker is shown in Figure 1.1. This model has a very irregular distribution of geometry with many degenerate triangles. Benchmark models that we use to demonstrate generality of our research are summarized at Table 1.1.

- **Massive models:** We would like to deal with massive models consisting of tens or hundreds of millions of triangles for interactive view-dependent rendering and collision detection.

- **High quality and accuracy:** It is very important to minimize rendering or collision detection error caused by the simplification method and, more importantly, guarantee the error to be within the user specified error bound. Our goal for dynamic simplification includes providing bounds on quality and accuracy of view-dependent rendered images and approximate collision detection.

Model	Type	Vert. (M)	Tri. (M)	Objs	Fig.
Bunny	s	0.03	0.06	1	Fig. 7.5
Dragon	s	0.4	0.8	1	Fig. 6.13
Turbine	s	0.9	1.7	1	Fig. 1.5
Lucy	s	14.0	28.0	1	Fig. 1.5
St. Matthew	s	186.0	372.0	1	Fig. 6.2
Power plant	c	11	12.2	1200	Fig. 4.9
Double eagle	c	77.7	81.7	3,346	Fig. 1.2
2M Isosurface	i	1	2	1	Fig. 3.6
Isosurface	i	50.5	100.0	N/A	Fig. 4.1
Puget sound	t	67.0	134.0	1	Fig. 6.6

Table 1.1: **Benchmark Models:** We use various kinds of models to demonstrate the generality of our research. The model types, model complexity, and figures revealing the rendered images of the models are shown. Type indicates model type: s for scanned model, i for isosurface, c for CAD model, and t for terrain model. **Vert.** is the number of vertices, **Tri.** is the number of triangles, and **Obj.** is the number of objects of a model. **Fig.** indicates a figure that shows the model.

- **Commodity hardware:** We want to design algorithms that work well on commodity hardware systems since commodity hardware is very easily accessible and affordable to many people.

1.4 Prior Work and Challenges

Dynamic simplification and cache-coherent layouts have been extensively researched. Dynamic simplification has been a subject of extensive research efforts, especially in regards to interactive view-dependent rendering. Also, cache-coherent layout has been widely researched in different fields of computer science. In this section, we briefly describe prior work and explain some of the issues with respect to our goal. More detailed work that relates our approaches are in Chapter 2.

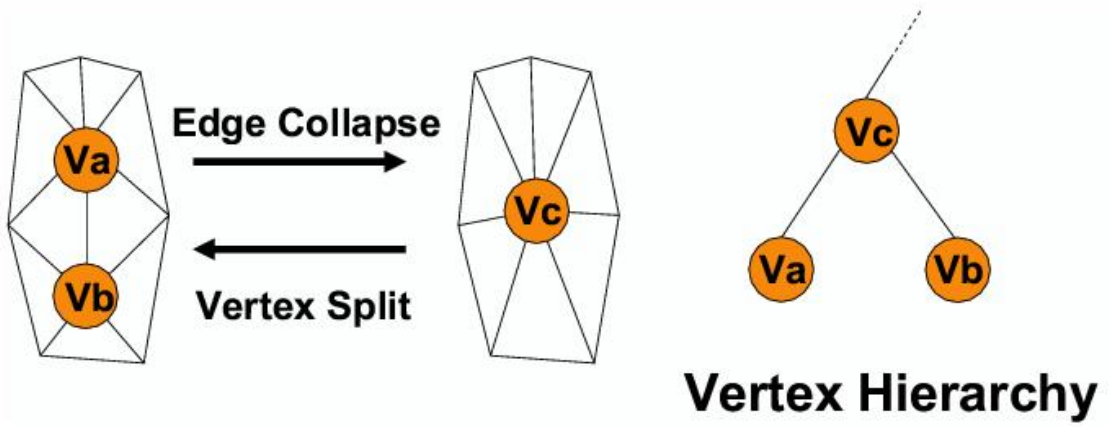


Figure 1.4: **An example of Simplification Operation and Vertex Hierarchy:** Edge collapse and vertex split on a simple mesh are shown at left. On the right, a vertex hierarchy constructed by the edge collapse is shown.

1.4.1 Dynamic Simplification

Most dynamic (or view-dependent) simplification algorithms encode an order of simplification operations such as edge collapses at preprocessing time. An appropriate LOD mesh is, then, chosen at runtime to meet the user-specified error bound according to view-dependent metrics. The order of simplification operations is usually organized as a hierarchy to efficiently provide view-dependent computation. The hierarchies for view-dependent computation can be classified as follows:

- **Vertex hierarchy:** The vertex hierarchy (Hoppe, 1997; Xia et al., 1997) is built from the original triangulated mesh. The interior nodes are generated by applying a simplification operation such as an edge collapse or vertex clustering to a set of vertices. The result of the operation is a new vertex that is the parent of the vertices to which the operator was applied. Successive simplification operations build a hierarchy that is either a single tree or a forest of trees. At runtime the mesh is refined to satisfy an error bound specified by the user. An example of a vertex hierarchy is shown in Fig. 1.4.

- **Half-edge hierarchy:** Pajarola (Pajarola, 2001) proposed a half-edge hierarchy for efficient view-dependent computation based on a half-edge collapse (Kobbelt et al., 1998) and a half-edge data structure (Weiler, 1985). Construction of the hierarchy is very similar to that of the vertex hierarchy. The primary advantage of the half-edge hierarchy over the vertex hierarchy is that it requires less memory. However, the half-edge hierarchy assumes that an input model is a manifold; its application to the real problems is limited.

- **Face hierarchy:** Floriani et al. (Floriani et al., 1997) proposed a *Multi Triangulation* (MT), which belongs to a class of face hierarchy. Each node contains a set of triangles. Triangles for intermediate nodes are constructed by performing simplification operations and re-triangulation on two sets of triangles of its child nodes. Its main advantage is that a face hierarchy has a direct relationship between simplified triangles; the face hierarchy has a more intuitive control on refinement and simplification; the vertex hierarchies, however, do not have such a benefit.

Each node of a hierarchy has preconditions (or *dependencies*) that each operation of refinement or simplification should meet. This precondition information is necessary to both guarantee the topological correctness of simplified meshes and to prevent any *fold-over*. This is because refining and simplification operations can be performed in an order that is different from the order of simplification operations obtained at preprocessing time. For example, each node of a vertex hierarchy has a set of triangles that should exist in the current LOD mesh before performing simplification or refining operations (Hoppe, 1997).

At runtime, a *front* of a hierarchy contains a set of nodes of the hierarchy. The front must be updated for every frame by determining whether nodes on the front should be replaced with their parent in order to decrease the level of detail, or replaced by their

children to increase the detail in a region of the mesh (Hoppe, 1997; Luebke & Erikson, 1997; Xia et al., 1997; Floriani et al., 1998).

Challenges and Issues: Many issues arise in applying these approaches to massive datasets composed of tens or hundreds of millions of triangles. We particularly discuss problems of view-dependent rendering based on vertex hierarchies since the vertex hierarchies are most widely used; similar issues are also applicable to other types of hierarchies. Issues of the vertex hierarchies can be classified as follows:

1. **Representation:** Each vertex split and edge collapse increases or decreases the number of triangles by two in most cases during traversal of a front of the vertex hierarchy. This very fine level granularity of modifying LODs creates smooth transitions when switching between different LODs. However, it can require a high number (e.g., 10K–100K) of LOD changes in each frame based on vertex splits and edge collapses for view-dependent rendering of massive models to meet an error bound specified by the user. Therefore, such a method will likely not achieve interactive performance on massive models.

2. **Construction:** Memory requirements of vertex hierarchies is very high; for example, Hoppe's view-dependent simplification based on the vertex hierarchy (Hoppe, 1997) reported 224 bytes for each vertex. If we extrapolate the memory requirement to a model consisting of 100 million vertices, 22.4GB is required to represent the model. Therefore, it is impractical to construct and load vertex hierarchies of massive models consisting of hundreds of millions of triangles with 32bit-based hardware, which typically has 1–2GB of main memory.

3. **Computation:** Traversing and refining a front across a vertex hierarchy composed of tens or hundreds of millions of polygons can take several seconds per

frame due to view-dependent computation and resolving dependencies. Moreover, dynamically generated geometry has low rendering performance on current graphics hardware. This is mainly caused by the low cache utilization during rendering of dynamically generated geometry (Luebke & Erikson, 1997).

4. **Integration with other acceleration techniques:** Out-of-core techniques are necessary to handle massive models that cannot fit into the main memory. However, resolving the dependencies can lead to non-localized memory accesses, which can be problematic for out-of-core management of the vertex hierarchy. Moreover, performing visibility culling and out-of-core management using vertex hierarchies can become expensive.

Moreover, only limited research has been done on using dynamic simplification on applications other than visualization due to issues mentioned above.

1.4.2 Cache-Coherent Layouts

One of the main characteristics of the current computing trend is the widening gap between the growth rate of the processing power of commodity hardware and slower growth rate of memory/disk access time and bandwidth. To bridge this widening gap, extensive research has been conducted on designing cache-coherent layouts of various data structures. These efforts can be classified as follows:

- **Graph layout:** Graph layout problems are categorized as combinatorial optimization problems. Their main goal is to find a linear layout of a graph such that a specific objective function is minimized. This work has been widely studied and an extensive survey is available (Diaz et al., 2002). Well known graph layout objective functions include the minimum linear arrangement (MLA) metric and

bandwidth. The MLA metric is the sum of index differences of vertices joined by the edges of the graph.

- **Sparse matrix reordering:** There is a considerable amount of research on converting sparse matrices into banded ones in order to improve the performance of various matrix operations (Diaz et al., 2002). Like graph layout, many sparse matrix reordering techniques compute a layout of a graph representing a sparse matrix such that an objective function is minimized.

- **Rendering and processing sequences:** The order in which a mesh is laid out affects the performance of rendering and performing computation on the mesh. In rendering applications, Deering (Deering, 1995) and Hoppe (Hoppe, 1999) showed how to take advantage of the existing vertex cache by reordering vertices and triangles of the mesh. In general streaming computation, Isenburg et al. (Isenburg et al., 2003) proposed processing sequences as an extension of rendering sequences to large-data processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices that can be streamed through main memory (Isenburg & Lindstrom, 2005).

- **Space filling curves:** Many algorithms use space filling curves (Sagan, 1994) to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve the performance of image processing (Velho & de Miranda Gomes, 1991) and terrain or volume visualization (Lindstrom & Pascucci, 2001; Pascucci & Frank, 2001). A standard method of constructing a layout is to embed the meshes or geometric objects in a uniform structure that contains the space filling curve.

Challenges and Issues: Although many approaches have been sought to reduce the access time of runtime applications, little attention has been directed to designing cache-

coherent layouts for a wide variety of applications including view-dependent rendering and collision detection between general and arbitrary polygon meshes.

Many graph layout and sparse matrix reordering techniques can be used for cache-coherent layouts in a wide variety of polygonal meshes. However, there is no direct relationship between these layout algorithms and minimizing access time in many geometric applications; the objective functions of graph layout and sparse matrix reordering do not capture access time or cache misses of runtime applications.

Also, rendering and processing sequences in computer graphics improve the performance of their computation by assuming that access patterns of runtime applications globally follow the fixed stored order of vertices and triangles. However, many applications including view-dependent rendering and collision detection have random, but cache-coherent access patterns, which cannot be predicted as a fixed order at preprocessing time.

Finally, space filling curves are widely known as heuristics, which reduce the access time of runtime applications. However, their applications are mostly evident on regular meshes, e.g. hight field, image, and volumetric grids, due to their geometric assumption of regularity on the underlying data. General and arbitrary polygonal meshes require further investigation because many data sets, especially CAD models, have very irregular geometric distribution. Also, some applications access their meshes based on topological relationships and not geometrical property.

1.5 Thesis Statement

We can design efficient and interactive view-dependent rendering and collision detection algorithms of complex and massive polygonal models by using dynamic simplification and cache-coherent layouts.

1.6 New Results

In this section we highlight major results of our research on dynamic simplification and cache-coherent layouts and their application to interactive visualization and collision detection.

1.6.1 Dynamic Simplifications for View-Dependent Rendering

We present two different dynamic simplification algorithms and their applications on interactive out-of-core view-dependent rendering for interactive display of massive models. The main new results of this thesis include:

1. **Model representation:** We use a novel representation, a *clustered hierarchy of progressive meshes* (CHPM) for dynamic simplification of massive models. The cluster hierarchy is used for coarse-grained view-dependent refinement in view-dependent rendering. Moreover, the cluster hierarchy is used for visibility computation and out-of-core management. The PMs provide fine-grained local refinement, which reduces the popping between successive frames without high refinement cost.

2. **Construction algorithms:** Our view-dependent rendering relies on an out-of-core construction algorithm to compute a CHPM that performs a cluster decomposition, generates a cluster hierarchy, and simplifies the original mesh by traversing the cluster hierarchy. We introduce the concept of *cluster dependencies* between adjacent clusters to generate drastic crack-free simplifications of the original model during the hierarchical simplification.

3. **Hybrid rendering algorithms:** Our algorithm uses the cluster hierarchy for visibility computation and out-of-core management in addition to view-dependent

rendering. We use temporal coherence and hardware accelerated occlusion queries for visibility computations at the cluster level. Our rendering algorithm introduces one frame of latency to allow newly visible clusters to be fetched without stalling the pipeline.

4. **Implementation and application:** We have implemented and tested our view-dependent rendering system on a PC with an NVIDIA 5950FX Ultra card. To illustrate the generality of our approach we have highlighted its performance on several models: a complex CAD environment (12M triangles), scanned models (372M triangles), and an isosurface (100M triangles). We can render these models at $15 - 35$ frames per second using a limited memory footprint of $400 - 600$MB.

1.6.2 Approximate Collision Detection

We present a fast and conservative collision detection algorithm for massive models composed of tens of millions of polygons. The main results of our algorithms can be classified as follows:

1. **Unified multiresolution representation:** We use the CHPM representation for error bounded collision detection and interactive visualization. Also, the CHPM representation serves as a *dual hierarchy* of each model. We use this representation both as a bounding volume hierarchy to cull away cluster pairs that are not in close proximity and as a multiresolution representation that adaptively computes a simplified representation of each model on the fly. Our algorithm utilizes the cluster hierarchy for coarse-grained refinement as well as progressive meshes (PMs) associated with each cluster for fine-grained local refinement. This allows us to rapidly compute a dynamic simplification and thereby reduce the "popping" or discontinuities between successive collision queries associated with

static levels of detail.

2. **Conservative error bound:** We also introduce a new conservative collision error metric. Based on this error metric, we compute the mesh simplification and perform overlap tests between the bounding volumes and the primitives. Our overall algorithm is conservative. It never misses any collisions between the original model, though it may return "false positive" collisions within an error bound.

3. **Fast and memory efficient GPU-based collision detection:** We use GPU-based occlusion queries for fast collision culling between dynamically-generated simplifications of the original models. Also, our algorithm requires less memory by using the CHPM representation. Moreover, we only load the cluster hierarchy into main memory and use out-of-core techniques to fetch the progressive meshes at runtime.

4. **Implementation and application:** Our algorithm has been implemented on a PC with an NVIDIA GeForce FX 5950 Ultra GPU and dual 2.5GHz Pentium IV processors. It has been used for real-time dynamic simulation between two complex scanned models consisting of 1.7M and 28M triangles and interactive navigation in a CAD environment composed of more than 12 million triangles. Image sequences of real-time dynamic simulation are shown in Figure 1.5. Collision queries using our algorithm take about $15 - 40$ milliseconds to compute all the contact regions on these benchmarks. Our system uses a memory footprint of approximately 250MB.

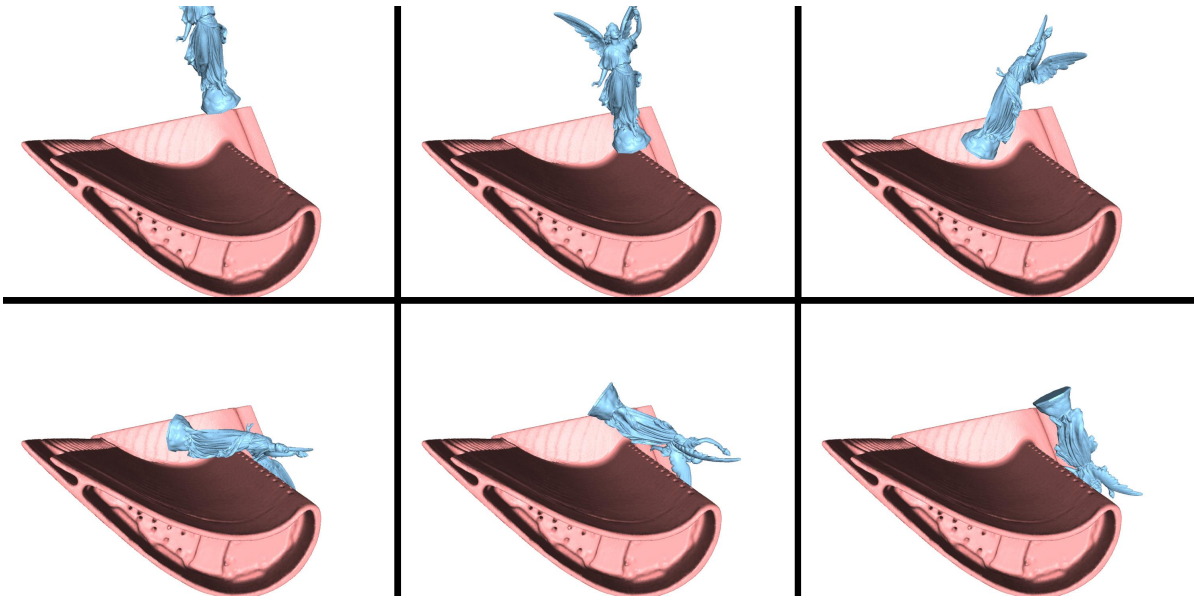


Figure 1.5: **Collision Example.** This image sequence shows discrete positions from our dynamic simulation application. The 28M-triangle Lucy model falls on and bounces off the 1.7M-triangle turbine-blade model and the response is computed using impulse-based simulation. In this simulation the collision detection took an average of 18ms per time step. The error bound, ϵ, was set to be 0.04% of the width of the Lucy.

1.6.3 Cache-Oblivious Layouts

We present novel methods to compute cache-oblivious layouts of polygonal meshes and hierarchies including bounding volume hierarchies and multiresolution meshes. Our approach is general, in that it allows for all types of polygonal models. It is also cache-oblivious in that it does not require any knowledge of the cache parameters or block sizes of the memory hierarchy involved in the computation. Our specific results include:

1. **Cache-oblivious metric:** We derive a practical and fast cache-oblivious metric that estimates the number of cache misses of runtime applications. Our metric assumes that all possible cache configurations are equally likely to happen at runtime and leads our layout algorithm to compute a new layout that can decrease the number of cache misses compared to a previous layout.

2. **Multilevel optimization:** We transform the layout computation to an optimization problem based on our metric and solve the combinatorial optimization problem using a multilevel minimization algorithm. The multilevel minimization enables us to efficiently compute cache-oblivious layouts of massive polygonal meshes.

3. **Natural extension to hierarchies:** Our layout algorithm is naturally applicable to cache-oblivious layout computations of hierarchies including bounding volume hierarchies and multiresolution hierarchies of large meshes. This is possible because our layout algorithm represents access patterns of runtime applications as a graph, whose nodes correspond to data elements and whose edges represent consecutive accesses between data elements that are likely to occur.

4. **Layout algorithm specialized to bounding volume hierarchies:** Bounding volume hierarchies are widely used to accelerate the performance of proximity

queries including collision detection and ray tracing. It is very important to construct high quality cache-coherent layouts of the hierarchies. To further improve cache-coherence of the bounding volume hierarchies, we propose a cache-oblivious layout algorithm optimized for bounding volume hierarchies by modeling access patterns on the hierarchies during proximity queries.

5. **Implementation and applications:** We have implemented our algorithm on a PC consisting of 2.4GHz Pentium-4 PC with 1GB of RAM and a GeForce Ultra FX 6800 graphics card. We have used cache-oblivious layouts for three different applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. In order to show the generality of our approach, we compute layouts of several kinds of geometric models. We use these layouts directly without any modification to the runtime application. Our layouts significantly reduce the number of cache misses and improve the overall performance. Compared to a variety of popular mesh layouts, we are able to achieve 2 to 20 times speedup in performance of view-dependent rendering, collision detection and isocontour extraction. Moreover, our layout algorithm for bounding volume hierarchies further improves the performance of collision detection over our general cache-oblivious layout algorithm.

1.7 Organization

The rest of the thesis is organized as follows:

- **Chapter 2** surveys the related work in the areas of dynamic simplification, collision detection, and cache-coherent algorithms in more detail.

- **Chapter 3** describes a vertex hierarchy augmented with a cluster hierarchy as a dynamic simplification representation in order to efficiently integrate view-

dependent rendering and visibility culling techniques.

- **Chapter 4** presents a novel dynamic simplification representation, the clustered hierarchy of progressive meshes (CHPMs), for interactive view-dependent rendering of massive and complex models. An out-of-core construction method for hierarchical simplification and an out-of-core visibility algorithm are also presented.

- **Chapter 5** investigates fast and approximate collision detection by using dynamic simplification. To quantify errors introduced by dynamic simplification, a conservative error metric is derived .

- **Chapter 6** proposes a novel algorithm that constructs cache-coherent layouts of polygonal meshes and hierarchies to minimize data access time of runtime applications including–but not limited to– view-dependent rendering and collision detection.

- **Chapter 7** investigates a cache-coherent layout algorithm further optimized for bounding volume hierarchies in order to improve performance of proximity queries including collision detection and ray tracing.

- **Chapter 8** suggests directions for future work and includes a conclusion of the thesis.

Chapter 2

Related Work

In this chapter, we will discuss prior work on dynamic simplification, cache-coherent algorithms, and collision detection, which are main focus of this thesis. There has been a considerable amount of research on each of these subjects; it is beyond the scope of this thesis to exhaustively survey the vast literature. We will discuss the details of related work that are most relevant to our work and highlight advantages and disadvantages of these algorithms.

2.1 Dynamic Simplification

Dynamic simplification or view-dependent simplification of polygonal models has been an active area of research over the last decade. We will first discuss the general mesh simplification method and, then, dynamic simplification methods for high-quality view-dependent rendering. We will also cover visibility culling and out-of-core techniques that have been used to improve the performance of dynamic simplification.

2.1.1 Mesh Simplification

Mesh simplification aims to reduce the polygon count of an input model while maintaining the quality of the input model as much as possible. By using the simplified

meshes, the rendering performance can be accelerated. This technique has been widely researched and an extensive survey is available (Luebke et al., 2002).

Most of the mesh simplification techniques have two major components: simplifying an input mesh based on atomic simplification operations and computing the error introduced by each simplification operation. At a high level, simplification operations can be classified as the following (Garland & Heckbert, 1997):

Vertex decimation: Each vertex is selected per every iteration of simplification (Schroeder et al., 1992). After the vertex is removed, re-triangulation is performed in order to fill the hole created by removing the vertex. One disadvantage of this approach is that re-triangulation inherently assumes manifoldness of the input mesh; therefore, it is not applicable to a wide variety of polygonal meshes.

Vertex clustering: A vertex clustering, as a simplification operation, selects a set of vertices and collapses them into a representative vertex. Rossignac and Borrel (Rossignac & Borrel, 1993) proposed a very fast simplification based on this vertex clustering. Initially, a uniform grid is overlaid on an input model and all the vertices within each cell of the grid are merged into a new vertex. The main advantages of this method is the fast performance and drastic simplification including topological changes. However, simplification quality obtained by this method is likely to be low and the size of the grid is not directly related to the geometric error bound of this simplification method.

Edge collapse: An edge consisting of two vertices are collapsed into a new vertex for simplification. Since only an edge collapses into a new vertex, triangulation after simplification can be incrementally computed. An example of edge collapse is shown in Fig. 1.4. Many methods (Hoppe, 1996; Hoppe, 1997) including ours are based

on edge collapses due to its simplicity. However, edge collapses can not merge two different objects into one if there are no edges between two objects; therefore, drastic simplification for many small objects cannot be achieved. To address this issue, virtual edges can be introduced by defining them of two vertices that are less than a particular distance threshold (Erikson & Manocha, 1999).

Once simplification operations are decided, error introduced by the simplification operations should be quantified. Two notable metrics related to our work are the error quadric metric (Garland & Heckbert, 1997) and the texture deviation metric (Cohen et al., 1998). Garland and Heckert introduced *error quadric* metric as a heuristic to measure the geometric error caused by edge collapse simplification operations. Intuitively speaking, an error quadric matrix measures the sum of the squared distance between a vertex and planes representing triangles of the mesh. A new vertex is positioned to minimize this error after an edge is collapsed. This error quadric matrix efficiently computes error caused by simplification and provides a high-quality simplification. We also use the quadric matrix method as our underlying error metric. Cohen et al. (Cohen et al., 1998) proposed a method that captures errors of geometry and appearance of the model–such as normal and color attributes–by using a texture deviation metric.

2.1.2 View-Dependent Rendering

View-dependent rendering using dynamic simplification originated as an extension of both the progressive mesh (PM) (Hoppe, 1996) and view-dependent metrics measuring projected geometric error in screen space (Lindstrom et al., 1996). A PM is a linear sequence of increasingly coarse meshes built from an input mesh by repeatedly applying edge collapse operations. It provides a continuous resolution representation of an input mesh and is useful for efficient storage, rendering and transmission. However, PMs are

not well suited for view-dependent rendering due to the nature of its linear sequences of LOD meshes stored in the PMs.

Xia and Varshney (Xia et al., 1997) and Hoppe (Hoppe, 1997) organized the PM as a vertex hierarchy (or view-dependent progressive mesh (VDPM)) instead of a linear sequence in order to address the shortcoming of PMs in view-dependent rendering. This representation allows a runtime application to take into account view-dependent effects such as silhouette preservation and lighting. Luebke and Erikson (Luebke & Erikson, 1997) developed a similar approach employing octree-based vertex clustering operations and used it for dynamic simplification. The Multi-Triangulation(MT) is a multiresolution representation that has been used for view-dependent rendering (Floriani et al., 1997; Floriani et al., 1998). All possible simplifications are explicitly represented in a MT. This property has been shown useful for perceptually guided simplification (Williams et al., 2003). Duchaineau et al. (Duchaineau et al., 1997) presented a view-dependent simplification algorithm for terrain models. Also, there have been several approaches to perform *geomorphs* between two different dynamic simplifications; this virtually removes any visual popping artifacts (Hoppe, 1997; Borgeat et al., 2005). These geomorphs can be integrated with our proposed methods.

Acceleration Techniques: Many techniques have been presented to improve the performance of view dependent rendering algorithms. El-Sana and Varshney (El-Sana & Varshney, 1999) used a uniform error metric based on cubic interpolants and reduced the cost of runtime tests. They also proposed *implicit dependencies* to improve refinement performance by reducing non-local memory access. Elsana et al. (El-Sana et al., 1999) applied skip lists to improve the rendering performance of view-dependent rendering. Pajarola (Pajarola, 2001) improved the update rate of runtime mesh selection by exploiting properties of the half-edge mesh representation and applied it to manifold objects. Bogomjakov and Gotsman (Bogomjakov & Gotsman, 2002) pre-

sented novel universal sequences to improve the rendering performance of progressive meshes. El-Sana and Bachmat (El-Sana & Bachmat, 2002) presented a mesh refinement prioritization scheme to improve the runtime performance. However, none of these aforementioned approaches have been demonstrated with complex and massive models consisting of tens or hundreds of millions of triangles.

2.1.3 Out-of-Core Simplification and Rendering

Many algorithms have been proposed for out-of-core simplification to handle massive models that cannot fit into main memory of commodity hardware. These include (Lindstrom, 2000; Lindstrom & Silva, 2001; Shaffer & Garland, 2001; Cignoni et al., 2003) for generating static LODs. Hoppe (Hoppe, 1998) extended the VDPM framework for terrain rendering by decomposing the terrain data into blocks, generating a block hierarchy and simplifying each block independently. Prince (Prince, 2000) extended this out-of-core terrain simplification to handle arbitrary polygonal models.

El-Sana and Chiang (El-Sana & Chiang, 2000) segment the input mesh into submeshes such that the boundary faces are preserved while performing edge-collapse operations. DeCoro and Pajarola (DeCoro & Pajarola, 2002) present an external data structure for the half-edge hierarchy and an explicit paging system for out-of-core management of view-dependent rendering. Lindstrom (Lindstrom, 2003) presents an end-to-end approach for out-of-core simplification and view-dependent visualization. Lindstrom's approach is based on memory insensitive simplification (Lindstrom & Silva, 2001) and has been applied to scanned models and isosurfaces. Although this out-of-core view-dependent approach was demonstrated on a massive model consisting of a few hundreds of millions of triangles, there is no guarantee that the original input mesh is correctly reconstructed when the highest resolution of the mesh is used at runtime.

Recently, Cignoni et al. (Cignoni et al., 2004) presented Adaptive TetraPuzzles

which builds a hierarchy of tetrahedrons and parallelized the computation of static LODs for nodes of the hierarchy. In Section 4.5 we compare our approach with the Adaptive TetraPuzzles algorithm.

2.1.4 Visibility Culling

The problem of computing the visible set of primitives from a viewpoint has been extensively studied in computer graphics and related areas. A recent survey of occlusion culling algorithms is given in (Cohen-Or et al., 2003). Occlusion culling algorithms may be classified as region or point-based, image or object space, and conservative or approximate.

Many occlusion culling algorithms have been designed for specialized environments, including architectural models based on cells and portals (Airey et al., 1990; Teller, 1992; Luebke & Georges, 1995) and urban datasets composed of large occluders (Coorg & Teller, 1997; Hudson et al., 1997; Schaufler et al., 2000; Wonka et al., 2000; Wonka et al., 2001). These approaches generally precompute a potentially visible set (PVS) for a region. However, these algorithms may not obtain significant culling on large environments composed of many small occluders.

Object space algorithms make use of spatial partitioning or bounding volume hierarchies (Coorg & Teller, 1997; Hudson et al., 1997); however, performing "occluder fusion" on scenes composed of small occluders with object space methods is difficult. Image space algorithms including the hierarchical Z-buffer (Greene et al., 1993; Greene, 2001) and hierarchical occlusion maps (Zhang et al., 1997) are generally more capable of capturing occluder fusion.

The PLP algorithm (Klosowski & Silva, 2000) uses an approximate occlusion culling approach that subdivides space into cells and assigns solidity values based on the triangles in each cell. This algorithm can provide a guaranteed frame rate at the expense of

non-conservative occlusion culling. However, it can lead to popping artifacts as objects can appear or disappear between successive frames. Klosowski and Silva (Klosowski & Silva, 2001) augment PLP with an image based occlusion test to design a conservative culling algorithm. The *iWalk* system (Correa et al., 2002) uses the PLP algorithm along with out-of-core preprocessing to render large models on commodity hardware.

A number of image-space visibility queries have been added by manufacturers to their graphics systems. These include the HP occlusion culling extensions, item buffer techniques, ATI's HyperZ hardware, and the NV_GL_occlusion_query OpenGL extension (Scott et al., 1998; Bartz et al., 1999; Greene, 2001; Klosowski & Silva, 2001; Hillesland et al., 2002; Meissner et al., 2002; Govindaraju et al., 2003c). Our integrated algorithm also utilizes these occlusion queries to perform occlusion culling.

Clustering: Often the original objects of a model are not represented in an optimal manner for occlusion culling algorithms. These algorithms need to represent the scene using an object hierarchy. Therefore, they create an object hierarchy by partitioning and clustering the model, and at runtime classifying objects as occluders and potential occludees. One recent approach to partitioning and clustering is presented by Baxter et al. (Baxter et al., 2002) and used in the GigaWalk system. It decomposes a large environment into geometrically almost equal-sized objects that are used for static LOD computations. Sillion (Sillion, 1994) and Garland et al. (Garland et al., 2001) presented hierarchical face clustering algorithms for radiosity and global illumination. These approaches are not directly applicable to generating a cluster hierarchy from a vertex hierarchy for view-dependent rendering and occlusion culling.

2.1.5 Hybrid Algorithm for Rendering Acceleration

Many hybrid algorithms have been proposed that combine model simplification with visibility culling or out-of-core data management. The Berkeley Walkthrough system (Funkhouser et al., 1996) combines cells and portals based on visibility computation algorithms with static LODs for architectural models. The MMR system (Aliaga et al., 1999) combines static LODs with occlusion culling and out-of-core computation and is applicable to models that can be partitioned into rectangular cells.

Other approaches combining precomputed static LODs and conservative occlusion culling have been proposed (Baxter et al., 2002; Govindaraju et al., 2003c; Govindaraju et al., 2003a). These algorithms represent the environment as a scene graph, precompute HLODs (hierarchical levels-of-detail) (Erikson et al., 2001) for intermediate nodes and use them for occlusion culling. However, switching between static LODs and HLODs can cause popping. Moreover, these algorithms use additional graphics processors to perform occlusion queries and introduce one frame of latency in the overall pipeline.

Wald et al. (Wald et al., 2004) combine out-of-core management with ray tracing and use volumetric approximation for unloaded geometry to improve out-of-core rendering performance. Govindaraju et al. (Govindaraju et al., 2003c) use hierarchies of static LODs (HLODs) and conservative occlusion culling for an interactive display of large CAD environments. El-Sana et al. (El-Sana et al., 2001) combined view-dependent rendering with approximate occlusion culling. The *iWalk* system (Corrêa et al., 2003; Corrêa, 2004) partitions the space into cells and performs out-of-core rendering of large architectural and CAD models on commodity hardware using approximate and conservative occlusion culling. However, there has been no approach combining dynamic simplification for view-dependent rendering and conservative visibility culling.

Point-based Rendering: An alternative rendering method to the traditional polygonal rendering is point-based rendering. The QSplat system (Rusinkiewicz & Levoy, 2000) uses a compact bounding volume hierarchy of spheres for view-frustum and back-face culling, levels of detail control and point-based rendering. It has been applied to large scanned models and works very well in practice. It is not clear whether point-based rendering algorithms would work on CAD models with sharp features or edges. Moreover, current graphics systems are well optimized to rasterize triangulated models. Recently, Dachsbacher et al. (Dachsbacher et al., 2003) exploited the programmability features of current GPUs to improve the rendering of performance of point primitives.

2.2 Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior (Vitter, 2001; Sen et al., 2002), and compiler optimizations based on tiling, strip-mining, and loop interchanging; all of these algorithms can reduce cache misses (Coleman & McKinley, 1995). Cache-efficient algorithms can be classified into the two standard techniques: computation reordering and data layout optimization.

2.2.1 Computation Reordering

Computation reordering is performed by computing a cache-coherent order of runtime computations; this is done in order to improve the program locality, that is, reduce the number of cache misses during runtime computations. This is typically performed using compiler optimizations or application specific hand-tuning.

At a high level, computation reordering methods can be classified as either cache-

aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size (Vitter, 2001). On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters (Frigo et al., 1999). There is a considerable amount of literature on developing cache-efficient computation reordering algorithms for specific problems and applications, including numerical programs, sorting, geometric computations, matrix multiplication, FFT, and graph algorithms. More details are given in recent surveys (Arge et al., 2004; Vitter, 2001).

Out-of-Core Mesh Processing: Out-of-core algorithms are designed to handle massive datasets on computers with finite memory. A recent survey of these algorithms and their applications is given in (Silva et al., 2002). The survey includes techniques for efficient disk layouts that reduce the number of disk accesses and the time taken to load the data required at runtime. Other algorithms use prefetching techniques based on spatial and temporal coherence. These algorithms have been used for model simplification (Cignoni et al., 2003), interactive display of large datasets composed of point primitives (Rusinkiewicz & Levoy, 2000) or polygons (Corrêa et al., 2003; Yoon et al., 2004b), model compression (Isenburg & Gumhold, 2003), and collision detection (Franquesa-Niubo & Brunet, 2003; Wilson et al., 1999).

2.2.2 Data Layout Optimization

The order of data elements of an underlying representation of a runtime application can have a major impact on the application's performance. For example, the order in which a mesh is laid out can affect the performance of algorithms operating on the mesh. Therefore, there have been considerable efforts on computing cache-coherent layouts of the data to match the runtime access pattern of applications. The following possibilities have been considered.

Graph Layouts: Graph layout problems are in the class of combinatorial optimization problem. Their main goal is to find a linear layout of an input graph such that a specific objective function is minimized. This work has been widely studied and an extensive survey is available (Diaz et al., 2002).

Well known graph layout problems includes a minimum linear arrangement (MLA) , which minimize the sum of index differences of two vertices consisting of an edge in the graph. The MLA problem is known to be NP-hard and its decision version is NP-complete(Garey et al., 1976). However its importance in many applications has inspired a wide variety of approximations based on heuristics including spectral sequencing (Juvan & Mohar, 1992), which minimizes the sum of squared index differences of edges. However, there has been no evidence that MLA or spectral sequencing of graphs can reduce the number of cache misses of runtime applications operating on the graphs.

Rendering Sequences: Modern GPUs maintain a small buffer to reuse recently accessed vertices. In order to maximize the benefits of vertex buffers for fast rendering, triangle reordering is necessary. This approach was pioneered by Deering (Deering, 1995). The resulting ordering of triangles is called a triangle strip or a rendering sequence. Hoppe (Hoppe, 1999) casts the triangle reordering as a discrete optimization problem with a cost function dependent on a specific vertex buffer size. If a triangle mesh is computed on the fly using view-dependent simplification or other geometric operations, the rendering sequences need to be recomputed to maintain high throughput. Other techniques improve the rendering performance of view-dependent algorithms by computing rendering sequences not tailored to a particular cache size (Bogomjakov & Gotsman, 2002; Karni et al., 2002). However, these algorithms have been applied only to relatively small models (e.g., 100K triangles).

Processing Sequences: Isenburg et al. (Isenburg et al., 2003) proposed processing sequences as an extension of rendering sequences to large-data processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices that can be streamed through main memory (Isenburg & Lindstrom, 2005). However, global mesh access is restricted to a fixed traversal order; only localized random access to the buffered part of the mesh is supported as it streams through memory. This representation is mostly useful for offline applications (e.g., simplification and compression) that can adapt their computations to the fixed ordering.

Space Filling Curves: Many algorithms use space filling curves (Sagan, 1994) to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing (Velho & de Miranda Gomes, 1991) and terrain or volume visualization (Lindstrom & Pascucci, 2001; Pascucci & Frank, 2001). A standard method of constructing a layout is to embed the meshes or geometric objects in a uniform structure that contains the space filling curve. Therefore, these algorithms have been used for objects or meshes with a regular structure (e.g. images and height fields). Methods based on space filling curves do not consider the topological structure of meshes. Our preliminary results indicate that these approaches do not work well with large CAD environments with an irregular distribution of geometric primitives. Moreover, if an application needs to access the mesh primitives based on connectivity information, space filling curves may not be useful. Algorithms have also been proposed to compute paths on constrained, unstructured graphs as well as to generate triangle strips and finite-element mesh layouts (Heber et al., 2000; Oliker et al., 2002; Bartholdi & Goldsman, 2004; Gopi & Eppstein, 2004).

Sparse Matrix Reordering: There is considerable research on converting sparse matrices into banded ones to improve the performance of various matrix operations

(Diaz et al., 2002). Common graph and matrix reordering algorithms attempt to minimize one of three measures: bandwidth (maximum edge length), profile (sum of maximum per-vertex edge length), and wavefront (maximum front size, as in stream processing). These measures are closely connected with MLA and layouts for streaming, and generally are more applicable to stream layout than cache-oblivious mesh layout.

Layouts of Bounding Volume Hierarchies: The impact of different layouts of tree structures has been widely studied. There is considerable work on cache-coherent layouts of tree-based representation. This includes work on accelerating search queries, which traverse the tree from the root node to descendant nodes. Given the cache parameters, Gil and Itai (Gil & Itai, 1999) casted computation of cache-coherent layouts given cache parameters as an optimization problem. They proposed a dynamic programming algorithm to minimize the number of cache misses during traversals of random search queries. However, there is much coherence on runtime traversals of proximity queries on BVHs; it is unclear that the technique will improve the performance of proximity queries. Recently, Alstrup (Alstrup et al., 2003) proposed a method to compute cache-oblivious layouts of search trees by recursively partitioning the trees.

There is relatively less work on cache-coherent layouts of BVHs. Opcode[1] used a blocking method that merges several bounding volumes nodes together to reduce the number of cache misses. The blocking is a specialized technique based on *van Emde Boas* layout of complete trees (van Emde Boas, 1977). The van Emde Boas layout is computed recursively. Given a complete tree, the tree is partitioned so that the height of the tree is divided into half. The resulting sub-trees are linearly stored by first placing the root sub-tree followed by other sub-trees from leftmost to rightmost. This process is applied recursively until it reaches a single node of the tree. However, it is

[1]http://www.codercorner.com/Opcode.htm

not clear whether the van Emde Boas layout minimizes the number of cache misses during traversal of BVHs, which may not be balanced or complete trees.

2.3 Collision Detection

The problem of collision detection has been well-studied in the literature. See recent surveys in (Jimenez et al., 2001; Lin & Manocha, 2003). Most of the commonly used techniques to accelerate collision detection between two objects utilize spatial data structures, including bounding volume and spatial partitioning hierarchies. Some of the commonly used bounding volume hierarchies (BVHs) include sphere-trees (Hubbard, 1993), AABB-trees (Beckmann et al., 1990), OBB-trees (Gottschalk et al., 1996), k-DOP-trees (Klosowski et al., 1998), etc. These representations are used to cull away portions of each object that are not in close proximity. A number of top-down and bottom-up methods have been proposed to build these hierarchies. Tan et al. (Tan et al., 1999) have used model simplification algorithms to generate tight fitting hierarchies.

Recently, GPU-based accelerated techniques have also been proposed for fast collision detection (Knott & Pai, 2003; Heidelberger et al., 2003; Govindaraju et al., 2003b; Kim et al., 2002). Their accuracy is governed by the frame-buffer or image-space resolution. Recently, Govindaraju et al. (Govindaraju et al., 2004) have presented a reliable GPU-based collision culling algorithm that overcomes these precision problems due to a limited frame-buffer resolution.

Dealing with massive models: There is relatively less work on collision detection between complex models composed of millions of polygons. The BVH based algorithms can be directly applied to these models. However, the memory overhead for the resulting algorithms can be substantial (e.g. many gigabytes). Wilson et al. (Wilson et al., 1999)

presented an out-of-core collision detection algorithm for large environments composed of multiple objects. Their algorithm uses spatial proximity relationships between different objects for out-of-core data management. Niubo and Brunet (Franquesa-Niubo & Brunet, 2003) have presented a K-dimensional data structure for broad-phase collision and proximity detection in large environments requiring external memory storage.

2.3.1 Approximate Collision Detection

In order to achieve interactive performance of collision detection between complex and massive models, many approximate algorithms have been proposed. Hubbard (Hubbard, 1993) introduced the concept of time-critical collision detection using sphere-trees. Collision queries can be performed as far down the sphere-trees as time permits, without traversing the entire hierarchy. This concept can be applied to any type of bounding volume hierarchy (BVH). However, no tight error bounds on collision results have been provided using this approach. O'Sullivan and Dingliana (O'Sullivan & Dingliana, 2001) studied LOD techniques for collision simulations and investigated different factors affecting collision perception, including eccentricity, separation, causality, and accuracy of simulation results. Otaduy and Lin (Otaduy & Lin, 2003) proposed CLODs, which are precomputed dual hierarchies of static LODs used for multiresolution collision detection. The runtime overhead of this approach is relatively small. However, switching LODs between successive instances may result in a large discontinuity in the simulation. Moreover, the underlying approach assumes that the input model is a closed, manifold solid and is not directly applicable to polygon soups.

Chapter 3

Dynamic Simplification integrated with Conservative Visibility Culling

Dynamic simplification based on vertex hierarchies has received considerable attentions since Lindstrom et al. (Lindstrom et al., 1996) used a vertex hierarchy specialzed for terrain models, Hoppe introduced view-dependent progressive meshes (VDPMs) (Hoppe, 1997), and other researchers introduced similar schemes (Xia et al., 1997; Luebke & Erikson, 1997). However, despite various advantages of dynamic simplification based on vertex hierarchies, the application of dynamic simplification to complex and massive models consisting of tens or hundreds of millions of triangles has been limited (See Section 1.4). For example, problems arise from traversing and refining a front, or cut, across the vertex hierarchy. In practice, refining a front for a model composed of hundreds of objects or millions of polygons can take several seconds or more per frame. Moreover, rendering the triangles in the front at interactive rates may not be possible, especially on models with high depth complexity.

Visibility culling techniques can be classified as conservative or approximate methods. Conservative visibility culling algorithms cull away portions of the scene that are not visible from the current view location using a potentially visible set (PVS). Most of these algorithms represent the scene using a spatial partition or bounding volume hierarchy and perform object-space or image-space culling tests to compute the PVS

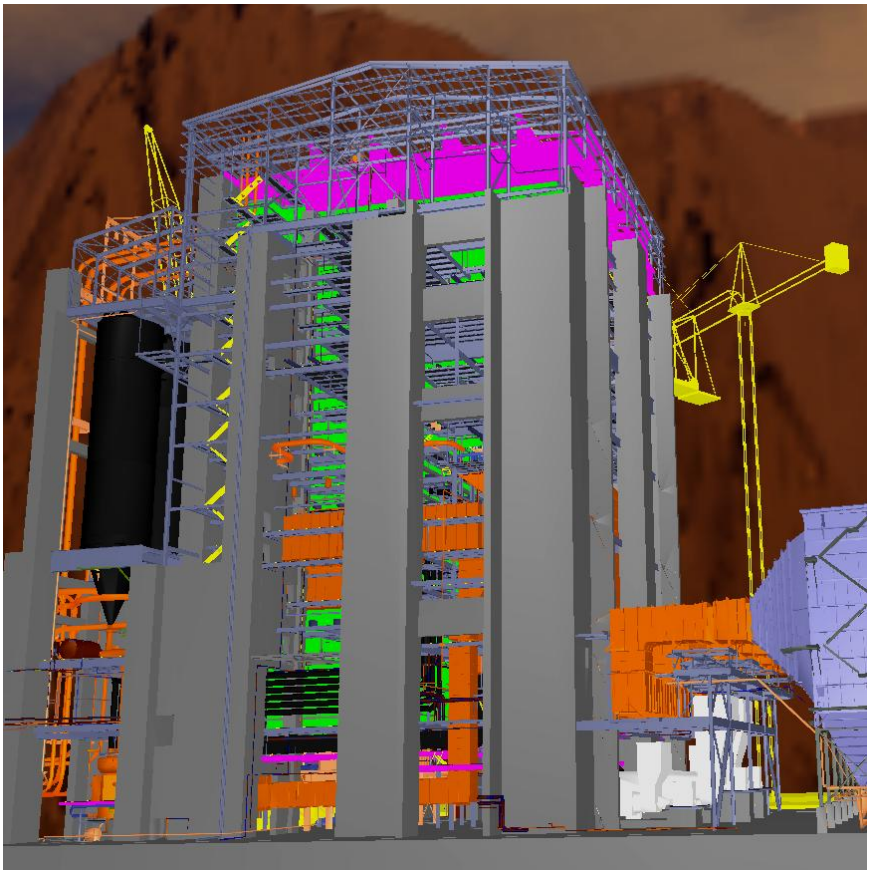


Figure 3.1: **Coal-Fired Power Plant:** This environment consists of over 12 million triangles and 1200 objects. Our view-dependent rendering with occlusion culling algorithm can render this environment at $10 - 20$ frames per second with very little loss in image quality on a Pentium IV PC with a NVIDIA GeForce 4 graphics card.

at runtime. On the other hand, approximate visibility culling may cull portions of the mesh that are visible to the viewer.

Given the complexity of large models and environments, integrated approaches that combine simplification and visibility culling are needed for interactive display. However, current techniques merely combine static LODs with conservative visibility culling or dynamic simplification with approximate visibility culling. Each of these techniques can generate popping artifacts at runtime.

In this chapter we investigate an integration method between dynamic simplification

based on vertex hierarchy and visibility culling to accelerate performance of high-quality view-dependent rendering.

Main Contribution: We propose an algorithm that combines dynamic simplification with conservative visibility culling for interactive view-dependent rendering. We precompute a vertex hierarchy of simplification operations for a large environment and a cluster hierarchy on top of the vertex hierarchy. We discuss a number of criteria to design a good cluster hierarchy and present techniques that automatically compute the hierarchy for large environments. We associate a bounding volume with each cluster so that the cluster hierarchy implicitly functions as a bounding volume hierarchy and is used to perform visibility culling using hardware accelerated visibility queries.

The runtime algorithm maintains a list of active clusters. This list is traversed as the mesh is refined within visible clusters to meet the error bound. The primitives within the refined clusters are rendered using vertex arrays. The cluster-based visibility culling algorithm limits the size of the active vertex front. As a result, the algorithm can refine and render the front at interactive rates.

The overall algorithm has been implemented on a Pentium IV PC with a NVIDIA GeForce 4 graphics card. It has been applied to two complex environments: a power plant model with more than 1200 objects and 12.2 million triangles, and an isosurface model composed of 2.4 million polygons[1] and a single object. The algorithm can render these datasets at $10 - 20$ frames a second with very little loss in image quality and minimal popping artifacts.

New Results: Some of the novel aspects of our work include:

[1]This model is now considered small given the high performance of current hardware. Moreover, this model can be interactively rendered without using any LODs. However, we believe that the technique described in this chapter will further improve the performance of rendering this model even in current comoodity hardware.

- **Integrated Scene Representation:** We propose an integrated scene representation for dynamic simplification and visibility computations based on a vertex hierarchy and a cluster hierarchy.

- **Clustering Algorithm for Vertex Hierachies:** An automatic cluster generation algorithm that takes into account several criteria important for visibility culling is described.

- **Dynamic Simplification Integrated with Conservative Visibility Culling:** To the best of our knowledge, our proposed view-dependent rendering system is based on the first integrated algorithm for dynamic simplification and conservative visibility culling that runs on commodity hardware, uses vertex arrays and is applicable to large and complex environments.

Organization: The rest of the chapter is organized as the following. In Section 3.1 we give a brief overview of our approach as well as the underlying representation. Section 3.2 describes the cluster hierarchy generation and partitioning algorithm. The runtime algorithm for view-dependent refinement and visibility culling is detailed in Section 3.3. We describe our implementation and highlight its performance on two complex environments in Section 3.4. Finally, in Section 3.5 we provide analsys of our approach and discuss some of its limitation. Portions of this chapter are described in (Yoon et al., 2003).

3.1 Overview

In this section we introduce some of the terminology and concepts used in our algorithm and give a brief overview of our approach.

3.1.1 Preprocess

Most view-dependent rendering algorithms use a vertex hierarchy built from an original triangulated mesh. The interior nodes are generated by applying a simplification operation such as an edge collapse or vertex clustering to a set of vertices. The result of the operation is a new vertex that is the parent of the vertices to which the operator was applied. Successive simplification operations build a hierarchy that is either a single tree or a forest of trees. At runtime the mesh is refined to satisfy an error bound specified by the user. Various issues on applying dynamic simplification based on the vertex hierarchies into massive models were discussed in Sec. 1.4.1.

We use the edge collapse operator as the basis for our vertex hierarchies and allow virtual edges so that disjoint parts of the model can be merged. We store an error value corresponding to the local Hausdorff distance from the original mesh with each vertex. This value is used to refine the mesh at runtime by projecting it to screen space where the deviation can be measured in pixels, which is referred to as "pixels of error."

A mesh "fold-over" occurs when a face normal flips during a vertex split or edge collapse. Vertex splits can be applied in a different order at runtime than during the hierarchy generation. This means that even though no fold overs occur during hierarchy generation, they may occur at runtime (Hoppe, 1997; Xia et al., 1997; El-Sana & Varshney, 1999). To detect this situation we use a neighborhood test. The face neighborhood is stored for each edge collapse and vertex split operation when creating the hierarchy. At runtime, an operation is considered fold-over safe only if its current neighborhood is identical to the stored neighborhood.

The vertex hierarchy can be interpreted as a fine-grained bounding volume hierarchy; each vertex of the hierarchy can have a bounding volume enclosing all faces adjacent. However, such a bounding volume hierarchy is not well suited for occlusion culling because each bounding volume is small and can occlude only a few primitives.

Furthermore, the culling algorithm will have to perform a very high number of occlusion tests.

To address this problem, we partition the vertex hierarchy into clusters and represent them as a cluster hierarchy. Each cluster contains a portion of the vertex hierarchy. All vertex relationships from the vertex hierarchy are preserved so that a vertex node may have a child or parent in another cluster. The relationships of the cluster hierarchy are based on those of the vertex hierarchy, so that at least one vertex in a parent cluster has a child vertex in a child cluster.

We characterize clusters based on their *error ratio* and *error range*. The error ratio is defined as the ratio of the maximum error value associated with a vertex in the cluster to that of the minimum. The error range is simply the range of error values between the maximum and minimum error values in a cluster. The error ratio and range are used in hierarchy construction, as described in Section 3.2.

We present a novel clustering algorithm that traverses the vertex hierarchy to create clusters that are used for occlusion culling.

3.1.2 Runtime Algorithm

In a standard VDR algorithm, the *front* (also referred to as the *active vertex list*) is composed of the vertices making up the current mesh representation. The front must be updated every frame by determining whether vertices on the front should be replaced with their parent to decrease the level of detail, or replaced by their children to increase the detail in a region (Hoppe, 1997; Luebke & Erikson, 1997; Xia et al., 1997). Additionally, a list of active faces, the *active face list* is maintained. In our algorithm the front is divided among the clusters. The active front will only pass through a subset of the cluster hierarchy which is called the "active clusters." These active clusters are traversed, and the active vertex front is refined within each active cluster. We do not

refine active clusters that are occluded, leading to a dramatic improvement in the front update rate and decreased rendering workload while still conservatively meeting the error bound.

Occlusion culling is performed by exploiting temporal coherence. During each frame, the set of clusters visible in the previous frame is used as an occluder set. These clusters are first refined by traversing their active fronts and then rendered to generate an occlusion representation. Next, the bounding volumes of clusters on the active front are tested for visibility. Only the visible clusters are refined and rendered using vertex arrays. This visible set then becomes the occluder set for the subsequent frame.

3.2 Clustering and Partitioning

In this section we present the cluster hierarchy generation algorithm. We initially describe some desirable properties of clusters for occlusion culling and present an algorithm designed with these properties in mind. We also present techniques to partition the vertices and faces among the clusters.

3.2.1 Clustering

We highlight some criteria used to generate the clusters from a vertex hierarchy, before describing our clustering algorithm. We have chosen oriented bounding boxes (OBBs) as our bounding volume because they can provide a tighter fit than spheres or axis aligned bounding boxes (Gottschalk et al., 1996). OBBs require more computation than simpler bounding volumes, but clustering is a preprocess that is performed once per environment.

Initially we consider issues in generating clusters that are not directly descended from each other; that is, they come from different branches of the cluster hierarchy.

Such clusters should have minimal overlap in their bounding volumes for two reasons. First, highly interpenetrating clusters are unlikely to occlude each other. Second, when rendering their bounding volumes, the required fill-rate is higher when they overlap. However, a parent cluster's bounding box should fully contain all the triangles and vertices of its children so that when it is deemed fully occluded, the subtree rooted at that cluster may be skipped. We also want to control the number of vertices and faces in a cluster so that we have uniformly sized occluders and occludees.

For occlusion culling it is desirable to have only one active cluster in a region of the mesh. If clusters have low error ratios, it is likely that multiple clusters will have to be active in a mesh region. This is possible since appropriate simplified LODs to meet an error bound can span those clusters. On the other hand, a cluster that has a high error ratio will contain vertices spanning many levels of the hierarchy in its mesh region. In this case, few of the vertices contained in a cluster will be active from any given viewpoint. Therefore, we must balance the error ratio of clusters. Also, the error range of a cluster should not overlap with its parent or children. Otherwise, it is likely that they will contain active vertices simultaneously.

These **properties** for the clusters can be summarized as:

1. Minimal overlap of bounding boxes of clusters not directly descended from each other.

2. Triangles and vertices contained in a cluster are fully contained within the bounding box of its parent cluster.

3. Minimal or no overlap of error range between parent and children clusters.

4. The error ratio is not too small or too large for a cluster.

5. The vertex and face count within a cluster are neither very large nor very small.

3.2.2 Cluster Hierarchy Generation

Our clustering algorithm works directly on an input vertex hierarchy without utilizing a spatial subdivision such as an octree. We assume that the vertex hierarchy from which the cluster hierarchy is generated exhibits high spatial coherence and is constructed in a bottom-up manner using edge collapses or vertex clusterings.

A cluster hierarchy can be generated by either using a bottom-up or top-down approach. A benefit of the bottom-up approach is spatial localization, but we assume that the vertex hierarchy already has this property. The top-down approach enables us to reduce the overlap of cluster bounding boxes. For this reason, we have chosen the top-down approach.

We descend the vertex hierarchy from the roots while creating clusters. An active vertex front is maintained and vertices on the front are added to clusters. When a vertex is added to a cluster, it is removed from the front and replaced with its children. We do not add a vertex to a cluster if it cannot be split in a fold-over safe manner. Thus, the construction of such a cluster will have to wait until dependent vertices are added to other clusters. For this reason, we use a cluster queue and place a cluster at the back of the queue when we attempt to add a vertex that is not fold-over safe. Then, the cluster at the front of queue is processed.

Each cluster in this cluster queue has an associated vertex priority queue sorted based on error values. A cluster's vertex queue contains its candidate vertices on the active front. Initially, the cluster queue contains a single cluster. The vertex priority queue associated with this initial cluster contains the roots of the vertex hierarchy. Since candidate vertices within a cluster are processed in order of decreasing error value[2], it is never the case that a vertex split is dependent upon a split in its own

[2]We guarantee that the error value of a node is bigger than the maximum error value of two child nodes.

vertex queue.

While the cluster queue is not empty the following steps are performed:

1. Dequeue the cluster, C, at the front of the cluster queue.

2. Dequeue the vertex, v, with highest error from the vertex priority queue.

3. If splitting v is not fold-over safe, return it to the vertex priority queue, place C at the back of the cluster queue and go back to Step 1.

4. If adding v to C makes the error ratio of C too large[3] or increases its vertex count beyond the target:

   (a) Create two children clusters C_l and C_r of C in the cluster queue.

   (b) Partition the vertex priority queue and assign the two resulting queues to C_l and C_r.

   (c) Go back to Step 1 without placing C in the back of the cluster queue; no more vertices will be added to this cluster.

5. Add v to C, update the number of vertices and the error ratio associated with C.

6. Replace v on the active vertex front by its children and enqueue the children in the vertex priority queue associated with C. Go back to Step 2.

This clustering algorithm ensures the properties highlighted in Section 3.2.1. Section 3.2.3 will explain how Property 1 is enforced when a cluster is partitioned. Property 3 is maintained by our algorithm as the vertices are inserted into the clusters from the vertex priority queue in order of decreasing error, so that children clusters always contain vertices with less associated error than their parent cluster. Properties 4 and

[3]When the error ratio of a cluster is too large, but its vertex count is too small (e.g., less than 10% of its target), we double the target error ratio in order to avoid too small cluster.

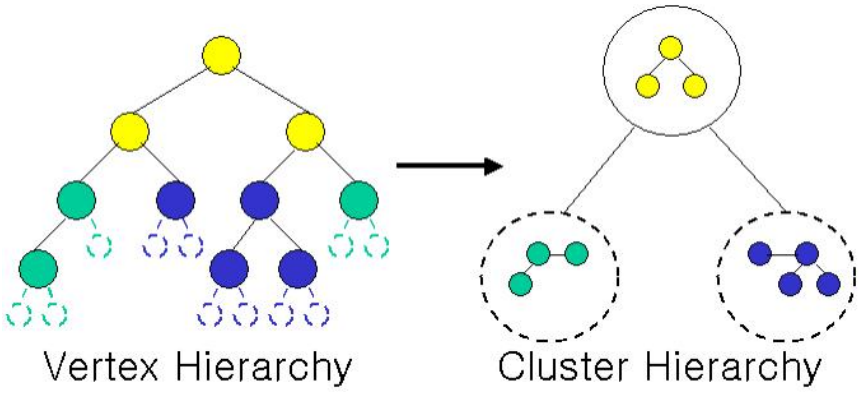


Figure 3.2: **Construction of the Cluster Hierarchy:** On the left is the input vertex hierarchy. The vertices are colored based on the cluster to which they are assigned. The nodes drawn with dotted lines represent the candidate vertices for the clusters, which reside in the vertex priority queue. The two clusters within dotted circles are still in the cluster queue, while the cluster inside the solid circle is finished processing.

5 cause the clusters to be split as the procedure traverses down the vertex hierarchy in Step 4.

Property 2 is enforced in a second pass after clustering by a bottom-up traversal which computes each parent cluster's bounding box by taking the union of its children. An example of a simple cluster hierarchy that is generated from vertex hierarchy is shown in Figure 3.2. Figure 3.3 shows the clusters on a bunny model at runtime.

3.2.3 Partitioning a Cluster

In Step 4(b) of the cluster generation algorithm, it is necessary to divide a cluster by splitting its vertex priority queue. The two resulting vertex priority queues form the initial vertex priority queues for the two children clusters.

To partition a cluster we compute a splitting plane for the vertices in the queue using principal component analysis. The eigenvector associated with the largest eigenvalue is initially used to define a splitting plane through the centroid of the vertices to maximally separate the geometry (Jolliffe, 1986). The vertices and associated faces are divided

Figure 3.3: **Clusters represented in a Vertex Hierarchy:** The clusters of the bunny model are shown in color. Clusters at 0 pixels of error are on the left and at 4 pixels of error are on the right.

based on this splitting plane, and an oriented bounding box is computed that contains the faces of each cluster. Bounding boxes are oriented with the splitting plane.

Some faces have a vertex in each of the newly created priority queues. As a result, their bounding boxes can overlap. This overlap can be very large when the cluster being split contains long, skinny triangles. Let V be the volume of the bounding box of the parent node and V_1 and V_2 be the volumes of the children bounding boxes. We use $(V_1 + V_2 - V)$ as a measure of the overlap of the children's bounding boxes. If this value exceeds a threshold fraction of V then the overlap is too large. In this case, the eigenvector corresponding to the second largest eigenvalue is used to define a new splitting plane. If this split again fails the overlap test, the third eigenvector is used. If all three fail, then we enforce Property 1 by abandoning the split and keeping the parent cluster in the cluster queue and increase either the target vertex count or the error ratio.

3.2.4 Memory Localization

After assigning vertices to clusters, we store the vertices in their corresponding clusters along with their associated faces. Performing this memory localization is useful for rendering using vertex arrays and on demand loading of clusters at runtime. Also, memory accesses when processing a cluster are more likely to be localized.

However, the vertices of a triangle can reside in different clusters. This is unavoidable in practice, no matter how the vertices are partitioned among different clusters. We deal with this situation by assigning each triangle to a single cluster containing at least one of its vertices. The cluster must store all three vertices of any triangle assigned to it, leading to some duplication of vertex data. Note, however, that only the data necessary to render such vertices is duplicated. The vertex hierarchy relationships are stored for each vertex only in the cluster to which they were assigned during cluster generation.

3.3 Interactive Display

In this section we present the runtime algorithm that uses the vertex and cluster hierarchy to update the active mesh for each frame and to perform occlusion culling. First, we present algorithms for model refinement followed by occlusion culling.

3.3.1 View-Dependent Model Refinement

In our algorithm the active vertex front or list and active face list, defined in Section 3.1.2, are divided among the clusters so that each cluster maintains its own portion of the active lists. Only clusters that contain vertices on the active front need to be considered during refining and rendering. These clusters are stored in an *active cluster list*. Figure 3.4 shows a cluster hierarchy, its active cluster list, and active vertex lists.

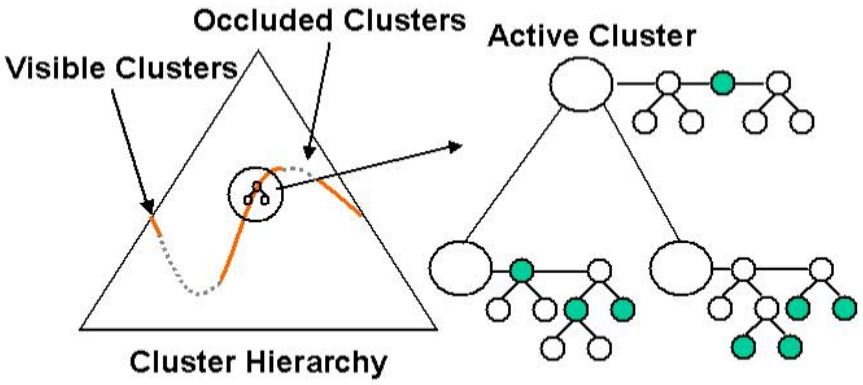


Figure 3.4: **Cluster Hierarchy and Vertex Hierarchy at Runtime:** The cluster hierarchy is used at runtime to perform occlusion culling. On the left, the active cluster list is drawn as a front across the cluster hierarchy. This list is composed of visible clusters and occluded clusters. Each cluster contains a portion of the vertex hierarchy as seen on the right. A subset of vertices in active clusters make up the current mesh. These are shaded on the right.

Prior to rendering a cluster, its active face and vertex lists are updated to reflect viewpoint changes since the last frame. We traverse its active vertex list and use the aforementioned vertex error value to compute which vertices need to be split or collapsed. The error value is projected onto the screen and used as a bound on the deviation of the surface in screen pixels. Vertex splits are performed recursively on front vertices that do not satisfy the bound. For sibling pairs that meet the error bound, we recursively check whether their parent vertex also meets the error bound and if so, collapse the edge (or virtual edge) between the vertex pair.

Faces in the active face list adjacent to a vertex involved in either an edge collapse or vertex split are replaced with faces adjacent to the new vertex. When a vertex is to be split, we use the neighborhood test to determine whether the vertex split is fold-over safe. However, vertex splits must occur to satisfy the error bound. To allow a split, we force any of its neighboring vertices to split when they are not part of the stored neighborhood as in (Hoppe, 1997).

3.3.2 Maintaining the Active Cluster List

A vertex that is split may have children that belong to a different cluster. The children vertices are activated in their containing clusters and these clusters are added to the active cluster list if they were not previously active. Similarly, during an edge collapse operation, the parent vertex is activated in its containing cluster and that cluster is added to the active cluster list. When the last vertex of a cluster is deactivated, the cluster is removed from the active cluster list.

3.3.3 Rendering Algorithm

Our rendering algorithm exploits frame-to-frame coherence in occlusion culling, by using the visible set of clusters from the previous frame as the *occluder set* for the current frame. The algorithm proceeds by rendering the occluder set to generate an occlusion representation in the depth-buffer. Then, it tests all the clusters in the active cluster list for occlusion. Meanwhile, the occluder set is updated for the next frame. An architecture of the runtime algorithm is shown in Figure 3.5. Different phases of the algorithm are numbered in the upper left of each box.

Occlusion Representation Generation

We use clusters that were visible in the previous frame for computing an occlusion representation. Before generating the representation, the active vertex list and active face list in each of these clusters are updated to meet the error bound. This refinement occurs as described in Section 3.3.1. This is Phase 1 of our algorithm. In Phase 2, the active faces are rendered and the resulting depth map is used as an occlusion representation.

Occlusion Tests

We traverse the active cluster list and cull clusters that are occluded or outside the view-frustum in Phase 3. The visibility of a cluster within the view frustum is computed by rendering its bounding box and then using a hardware occlusion query to determine whether any fragments passed the depth test. Depth writes are disabled during this operation to ensure that the bounding boxes are not used as occluders. Also, depth clamping is enabled so that we do not need to consider special case bounding boxes that are intersecting the near clip plane. The active vertex front may pass through a cluster and some of its descendant clusters. Since the bounding box of a cluster fully contains the bounding boxes of its children, once a cluster is found to be occluded we do not have to check its children.

During this phase, all the clusters in the active cluster list are tested, including those in the occluder set. This test is necessary because the clusters that pass the visibility test are used as occluders for the subsequent frame. In this manner, clusters that become occluded are removed from the occluder set.

Refining Visible Clusters

The previous phase allows us to determine which clusters are potentially visible. Before rendering the potentially visible clusters in Phase 5, their active face and vertex lists must be updated in Phase 4. While refining, additional clusters may be added to the active cluster list through vertex splits and edge collapses. These clusters are assumed to be visible in the current frame.

3.3.4 Conservative Occlusion Culling

The bounding box test conservatively determines whether the geometry within a cluster will be occluded, since a bounding box contains all the faces associated with a cluster.

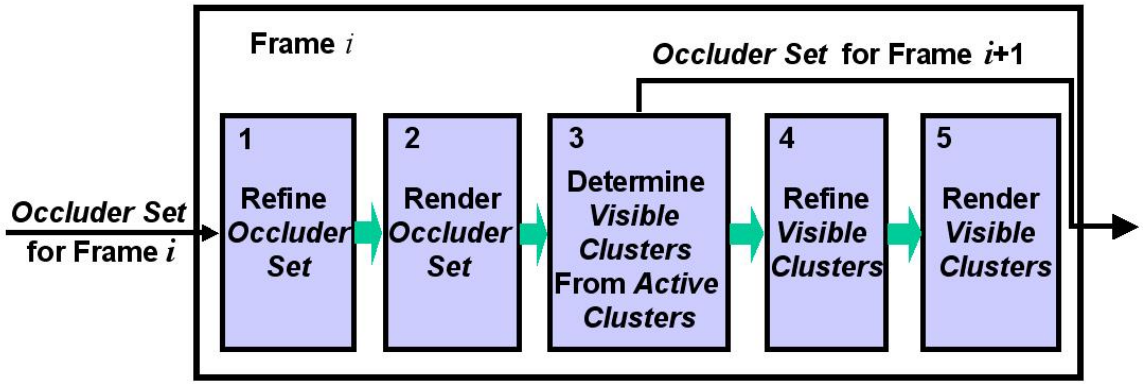


Figure 3.5: **Runtime System Architecture:** In each frame the clusters visible in the previous frame are used as an occluder set. In Phases 1 and 2, the occluder set is refined and then rendered to create a depth map in the z-buffer. Phase 3 tests bounding boxes of all the active clusters against this depth map using occlusion queries. The clusters passing the test are refined and rendered in Phases 4 and 5 and also used as occluders for the next frame.

We also ensure conservativeness up to screen-space precision by refining the occluder set in Phase 1 before generating the depth map in Phase 2.

To prevent refining and rendering the same cluster two times during a frame, the occluder set rendered in Phase 2 is also rendered into the color buffer. Then, when refining and rendering the visible clusters in Phases 4 and 5, we omit the clusters that were already refined and rendered in Phases 1 and 2. This optimization requires an extra step to ensure conservativeness.

As explained in Section 3.3.1, the neighborhood vertices may be forced to split to satisfy the error bound. A problem arises when a vertex split in Phase 4 forces a vertex in a cluster already rendered in Phase 2 to split. We detect such cases and redraw the resulting faces, so that no visual artifacts remain in the final image. To achieve this, we first rerender the affected faces prior to the split into the stencil buffer after setting the depth function to GL_EQUAL. After the split, the correct faces are rendered and overwrite pixels only where the stencil has been set. We have found that this occurs

very rarely (on average less than one face per frame in our datasets).

3.3.5 Vertex Arrays

On current graphics processors display lists and vertex arrays are significantly faster than immediate mode rendering (Woo et al., 1997). The changing nature of the visible primitives and dynamically generated LODs in a VDR system are not well suited for display lists. Thus, we use vertex arrays stored in the graphics processor unit (GPU) memory to accelerate the rendering.

We use a memory manager when the size of the vertices in the active clusters is less than the amount of the memory allocated on the GPU (e.g. 100 MB). Using a least recently used replacement policy, we keep the vertices in GPU memory over successive frames. When the front size exceeds the memory requirement, we still use GPU memory, but do not attempt to keep clusters in this memory for more than one frame.

In many rendering applications all or most of the vertices in a vertex array are used to render faces. But in our case only a fraction of the vertices for a cluster, the active vertices, are used for rendering. This increases the number of bytes per rendered vertex that are transferred to the GPU when using vertex arrays stored in GPU memory. To obtain maximum throughput, we use a minimum ratio of active vertices to total vertices, and any active cluster that does not meet this threshold is rendered in immediate mode.

3.4 Implementation and Results

In this section we discuss some of the details of our implementation and highlight its performance on two complex environments.

Model	**Poly**$\times 10^6$	**Obj**	**Cluster**$\times 10^3$
2M Isosurface model	2.4	1	1.3
Power plant	12.2	1200	20.1

Table 3.1: **Details of Test Environments: Poly** is the polygon count. The **Obj** column lists the number of objects in the original dataset and the **Cluster** column lists number of clusters generated.

3.4.1 Implementation

We have implemented our view-dependent rendering algorithm with conservative occlusion culling on a 2.8 GHz Pentium-IV PC, with 4 GB of RAM and a GeForce 4 Ti 4600 graphics card. It runs Linux 2.4 with the bigmem option enabled giving 3.0 GB user addressable memory. Using the NVIDIA OpenGL extension GL_NV_occlusion_query, we are able to perform an average of approximately 100K occlusion queries per second on the bounding boxes.

For higher performance, we allocate 100MB of the 128MB of RAM on the GPU to store the cluster vertices and bounding boxes. The memory allocated on the graphics card can hold about 3.5 million vertices.

3.4.2 Environments

Our algorithm has been applied to two complex environments, a coal fired power plant composed of more than 12 million polygons and 1200 objects (shown in Fig. 3.1) and an isosurface model consisting of 2.4 million polygons and a single object (shown in Fig. 3.6). The details of these environments are shown in Table 3.1.

We use GAPS (Erikson & Manocha, 1999) to construct our vertex hierarchies because it handles non-manifold geometry and can also perform topological simplification. Because the GAPS algorithm requires large amounts of memory, we built hierarchies for portions of each environment separately and merged the results to compute a single

vertex and cluster hierarchy. A target of 1000 vertices is used while generating the clusters. The maximum error value of any vertex in the cluster is twice that of the minimum; that is, the error ratio is 2.

Our approach is designed for complex environments consisting of tens of millions of polygons. Partial loading can be very useful in such an environment. We decouple the vertex and face data from the edge collapse hierarchy stored in each cluster as described in Section 3.2.4. We do not load the face and vertex data for a cluster until it needs to be rendered. In this manner, clusters that never fall within the view-frustum or are always occluded will never be loaded when performing a walkthrough.

Preprocessing Time and Memory Requirements

Our cluster hierarchy generation algorithm can process about 1M vertices in 3.8 minutes. Almost 18% of that time is spent calculating the eigenvectors computed for principal component analysis when splitting clusters and determining OBBs. We optionally employ a step that attempts to tighten the OBBs by minimizing their volume while still enclosing the clusters. When this step is used, the time spent in cluster generation increases by ten times; the bounding box computation accounts for 90% of the time spent in the clustering step. We performed the minimization step during cluster generation for the power plant model and not for the isosurface model.

Our current implementation is not optimized in terms of memory requirements. Each cluster uses 300 bytes to store the bounding box information and other data. Each vertex and face has a 4 byte pointer indicating its containing cluster along with the geometric data. On average, we use 272Mb for 1M vertices. This number is slightly higher in comparison with some earlier systems for view-dependent rendering. For example, Hoppe's view-dependent simplification system (Hoppe, 1997) reported 224Mb for 1M vertices. The difference partly exists because our implementation supports

virtual edges and non-manifold topology, which means some relationships cannot be stored implicitly.

3.4.3 Optimizations

We use a number of optimizations to improve the performance of our algorithms.

Conservative Projected Error

When traversing the active vertex list of a cluster we use a conservative approximation of the distance from a vertex to the viewpoint. The minimum distance between a sphere surrounding a cluster and the viewpoint is computed. Then, the maximum surface deviation meeting the screen space error bound at this distance is calculated and all active vertices in the cluster are refined using this value. This approximation is conservative and requires only one comparison per vertex to determine whether it needs to be split or collapsed.

Multiple Occlusion Queries

The GL_NV_occlusion_query extension supported on the GeForce 3 and all subsequent NVIDIA GPUs allows many queries to be performed simultaneously. To get the result of a query, all rasterization prior to issuing the query must be completed. Thus, we wait until we have rendered all the bounding boxes in the active cluster list before gathering query results from the GPU.

3.4.4 Results

We generated paths in each of our environments and used them to test the performance of our algorithm. We are able to render both these models at interactive rates ($10 - 20$ frames per second) on a single PC.

Figure 3.6: 2**M Isosurface Model acquired from Turbulence Simulation:** This environment consists of 2.4 million triangles and is rendered by our system at interactive rates.

We have also compared the performance of our system to VDR without occlusion culling. We accomplish this comparison by disabling occlusion culling in our system, which involves simply refining and rendering all the clusters in the active cluster list. Moreover, we do not use the conservative approximation of the error distance, since this optimization is possible because of clustering used for occlusion culling. We use view-frustum culling, vertex arrays, and GPU memory to accelerate the rendering of the scene in each case. Figure 3.8 illustrates the performance of the system on a complex path in the power plant and isosurface model. Notice that we are able to obtain a $3-5$ times speedup with conservative occlusion culling. Table 3.2 shows the average frame

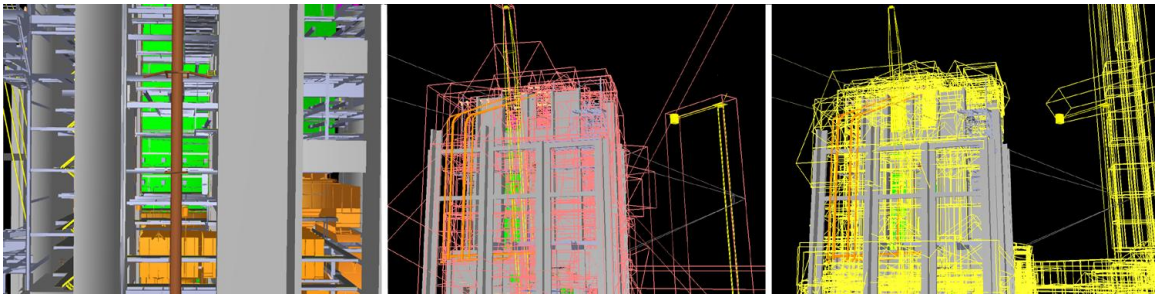


Figure 3.7: **Visibility Culling in the Power Plant:** The left image shows a first person view. The middle image shows a third person view with the bounding boxes of visible clusters shown in pink and the view frustum in white. The right image is from the same third person view with the bounding boxes of occluded clusters in yellow.

Model	Pixels of Error	FPS		Front Verts (K)		Merge/Split	
		VDR	VDR+OC	VDR	VDR+OC	VDR	VDR+OC
Iso.	0.5	6.4	19.7	195	113	2356	1222
PP	3	2.62	12.3	297	126	1973	559

Model	Poly (K)		Visible clusters in VDR+OC	VF culled clusters in VDR+OC	OC culled clusters in VDR+OC
	VDR	VDR+OC			
Iso.	311	224	349	106	299
PP	433	162	1166	390	1852

Table 3.2: **Runtime Performance:** Average frame rates and average number of split and merge operations obtained by different acceleration techniques are shown over the sample path. This result is acquired at 512×512 image resolution. **Iso.** = Isosurface model, **FPS** = Frames Per Second, **Poly** = Polygon Count, **PP** = Power Plant model, **VDR** = View-dependent Rendering with view frustum culling, **VF** = View Frustum, **OC** = Occlusion Culling

rate, front size, and number of edge collapse and vertex split operations performed during the path. The main benefit of occlusion culling arises from the reduction in the size of the front (by a factor of one third to one half) as well as the number of rendered polygons. Tables 3.3 and 3.4 show a breakdown of the time spent on the major tasks (per frame) in our system. Due to occlusion culling, the resulting front size and the time spent in refining the front is considerably smaller and yields improved performance. Note that our improvement in refining is even more dramatic than the

Step	Refining	Rendering	Culling
VDR+OC	17ms (34%)	20ms (38%)	14ms (28%)
VDR	136ms (81%)	31ms (19%)	–

Table 3.3: **Breakdown of Frame Time in 2M Isosurface Model:** Left values in each cell represent time spent in each step. Right values represent percentage of total frame time. The **Refining** column represents Phase 1 and 4, **Rendering** is Phase 2 and 5, and **Culling** is Phase 3.

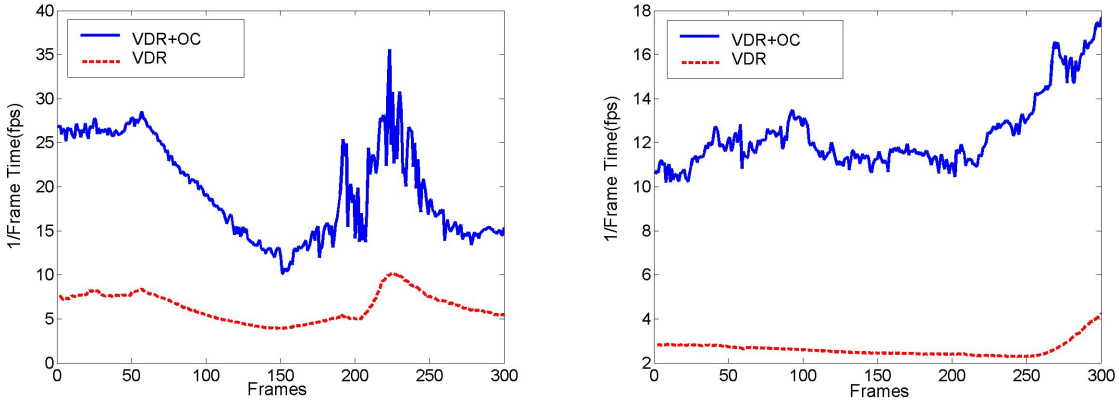
Step	Refining	Rendering	Culling
VDR+OC	23ms (28%)	27ms (33%)	31ms (39%)
VDR	213ms (56%)	169ms (44%)	–

Table 3.4: **Breakdown of Frame Time in Power Plant:** The columns **Refining**, **Rendering**, and **Culling** are explained in Table 3.3

improvement in rendering due to the conservative distance computation. Figure 3.7 shows visible and invisible clusters in a given viewpoint on the power plant model.

3.5 Analysis and Limitation

We have presented a novel algorithm for integrating dynamic simplification based on a vertex hierarchy with conservative visibility culling for interactive view-dependent rendering. Our algorithm performs clustering and partitioning to decompose a vertex hierarchy of the entire scene into a cluster hierarchy, which is used for view-frustum and visibility culling. At runtime, a potentially visible set of clusters is maintained using hardware accelerated occlusion queries, and this set is refined in each frame. The cluster hierarchy is also used to update the active vertex front that is traversed for view-dependent refinement. Our algorithm easily allows the use of vertex arrays to achieve high triangle throughput on modern graphics cards. We have observed $3-5$ times improvement in frame rate over view-dependent rendering without occlusion culling on two complex environments.



(a) Isosurface model at 0.5 pixel of error

(b) Power plant model at 3 pixels of error

Figure 3.8: **Frame Rate with/without Visibility Culling:** Frame rate comparison between VDR with and without occlusion culling. Image resolution is 512×512. We obtain a $3-5$ times improvement in the frame rate when using occlusion culling.

Comparison with Earlier Approaches: To the best of our knowledge, none of the earlier algorithms can perform dynamic simplification with conservative visibility culling for interactive view-dependent rendering. The iWalk system (Correa et al., 2002) can also render the power plant model on a single PC with much smaller preprocessing and memory overhead than ours. However, it does not use LODs and performs approximate and non-conservative occlusion culling. The GigaWalk (Baxter et al., 2002) and occlusion-switch algorithms (Govindaraju et al., 2003c) use static LODs with visibility culling. Although they can render the power plant model at interactive rates, they can produce popping due to switching between different LODs. Furthermore, they use more than one graphics processor.

An integrated algorithm combining view-dependent rendering with PLP-based approximate occlusion culling is presented in (El-Sana et al., 2001). Finally, (El-Sana & Bachmat, 2002) have presented a scheme for subdividing the vertex hierarchy at runtime to generate a coarser hierarchy. The cells of this hierarchy are split and merged to reflect the changes in the active front of vertices. These cells are prioritized by an esti-

mate of the number of vertex splits and edge collapses required in each cell. Refinement occurs over a subset of the active cells in each frame, considering the priority as well as ensuring that all cells are eventually refined. Our algorithm follows the same theme of reducing the front size and therefore, subdivides the vertex hierarchy into clusters as a preprocess. As a result, our algorithm is applicable to very large environments and the resulting clusters are used for visibility culling.

Limitations: Our visibility culling algorithm assumes high temporal coherence between successive frames. If the camera position changes significantly from one frame to the next, the visible primitives from the previous frame may not be a good approximation of the occluder set for the current frame. As a result, the culling performance may suffer. Furthermore, if a scene has very little or no occlusion, the additional overhead of performing occlusion queries can lower the frame rate.

Our algorithm performs culling at a cluster level and does not check the visibility of each triangle. As a result, its performance can vary based on how the clusters are generated and represented.

Chapter 4

Dynamic Simplification based on CHPM Representation

In earlier chapter we introduced a cluster hierarchy combined with a vertex hiearchy to provide visibility culling for dynamic simplification. Although we were able to improve performance of view-dependent rendering by enabling visibility culling by using the cluster hierarchy, underlying dynamic simplification representation is still vertex hierarchy. Therefore, our preivous representation inherits all the issues of vertex hierarchies; high refinement cost, high memory requirement, complicated integration with out-of-core management, and low rendering performance.

In this chapter, we present a new view-dependent rendering algorithm (Quick-VDR) for interactive display of massive models based on a novel dynamic simplification representation, a cluster hierarchy of progressive meshes. Main results of this chapter can be classified as the following:

1. **Model representation:** We use a novel scene representation, a *clustered hierarchy of progressive meshes* (CHPM). The cluster hierarchy is used for coarse-grained view-dependent refinement. The PMs provide fine-grained local refinement to reduce the popping between successive frames without high refinement cost.

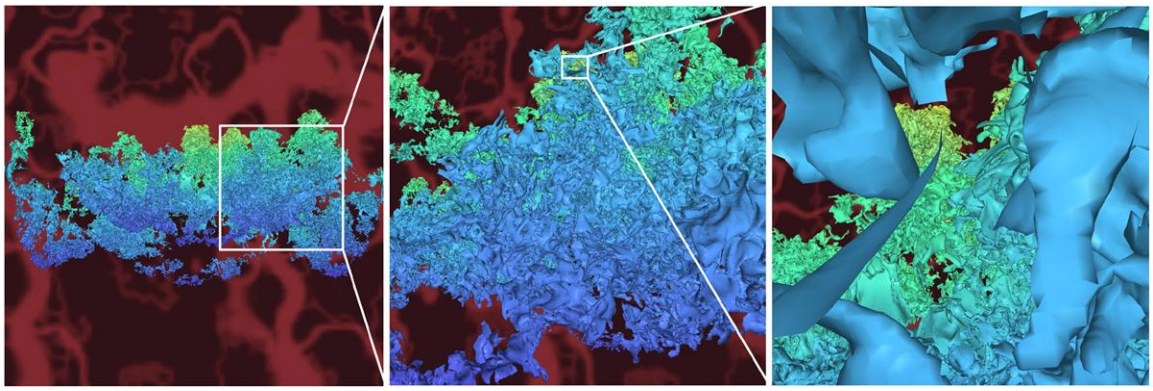


Figure 4.1: **Isosurface Model:** These images show the application of Quick-VDR to a complex isosurface (100M triangles) generated from a very high resolution 3D simulation of Richtmyer-Meshkov instability and turbulence mixing. The middle and right images show zoomed views. The isosurface has high depth complexity and many holes. Quick-VDR can render it at $15 - 40$ frames per second on a PC with NVIDIA GeForce 5950FX Ultra card and uses a memory footprint of 600MB.

2. **Construction algorithm:** Quick-VDR relies on an out-of-core algorithm to compute a CHPM that performs a hierarchical cluster decomposition and simplification. We introduce the concept of *cluster dependencies* between adjacent clusters to generate drastic crack-free simplifications of the original model.

3. **Rendering algorithm:** Our rendering algorithm uses temporal coherence and occlusion queries for visibility computations at the cluster level. We account for visibility events between successive frames by combining fetching and prefetching techniques for out-of-core rendering. Our rendering algorithm introduces one frame of latency to allow newly visible clusters to be fetched without stalling the pipeline.

4. **Implementation and Application:** We have implemented and tested Quick-VDR on a commodity PC with NVIDIA 5950FX Ultra card. To illustrate the generality of our approach we have highlighted its performance on several models: a complex CAD environment (12M triangles), scanned models (372M triangles),

and an isosurface (100M triangles). We can render these models at $15-35$ frames per second using a limited memory footprint of $400-600$MB.

Advantages: Our approach integrates view-dependent simplification, conservative visibility culling, and out-of-core rendering for high quality interactive display of massive models on current graphics systems. As compared to prior approaches, Quick-VDR offers the following benefits:

1. **Lower refinement cost:** The overhead of view-dependent refinement in the CHPM is one to two orders of magnitude lower than vertex hierarchies for large models.

2. **Massive models:** We are able to compute drastic simplifications of massive models, using hierarchical simplification with cluster dependencies, necessary for interactive rendering.

3. **Runtime performance:** Quick-VDR renders CHPMs using a bounded memory footprint and exploits the features of current graphics processors to obtain a high frame rate.

4. **Rendering quality:** We significantly improve the frame rate with little loss in image quality and alleviate popping artifacts between successive frames.

5. **Generality:** Quick-VDR is a general algorithm and applicable to all types of polygonal models, including CAD, scanned, and isosurfaces.

Organization: The rest of the chapter is organized in the following manner. We give a brief overview of our scene representation and refinement algorithm in Section 4.1. Section 4.2 describes our out-of-core algorithm to generate a CHPM for a large environment. We present the rendering algorithm in Section 4.3 and highlight its

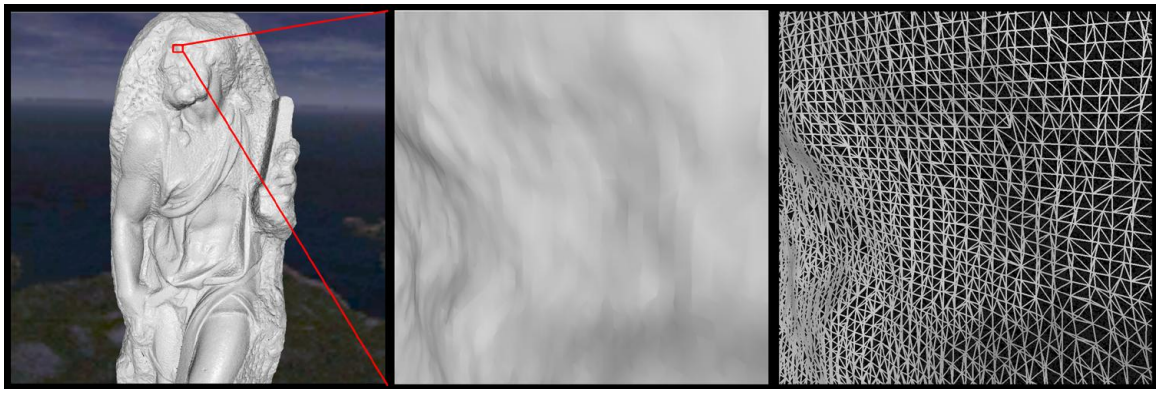


Figure 4.2: **Scan of Michelangelo's St. Matthew:** The statue was scanned with a sample spacing of .29mm and 0.1mm depth resolution. This 9.6GB scanned model consists of 372M triangles. The middle image is a zoomed view and the right image shows its triangulation. Quick-VDR is able to render this model at $13-25$ frames per second on a dual Pentium IV PC with a GeForce 5950FX Ultra GPU using a memory footprint of 600MB.

performance in Section 4.4. We compare our algorithm with other approaches in Section 4.5 and discuss some of its limitations. Portions of this chapter are described in (Yoon et al., 2004b; Yoon et al., 2005b).

4.1 Overview

In this section we introduce some of the terminology and representations used by Quick-VDR. We also give a brief overview of our approach for out-of-core hierarchical simplification and rendering.

4.1.1 Scene Representation

We propose a novel representation, a clustered hierarchy of progressive meshes (CHPM), for view-dependent rendering of massive datasets. The CHPM consists of two parts:

Cluster Hierarchy: We represent the entire dataset as a hierarchy of clusters, which are spatially localized mesh regions. Each cluster consists of a few thousand triangles. The clusters provide the capability to perform coarse-grained view-dependent (or selective) refinement of the model. They are also used for visibility computations and out-of-core rendering.

Progressive Mesh: We precompute a simplification of each cluster and represent linear sequence of edge collapses as a progressive mesh (PM). The PMs are used for fine-grained local refinement and to compute an error-bounded simplification of each cluster at runtime.

We refine the CHPM at two levels. First we perform a coarse-grained refinement at the cluster level. Next we refine the PMs of the selected clusters. The PM refinement provides smooth LOD transitions.

Cluster Hierarchy

Conceptually, a cluster hierarchy is similar to a vertex hierarchy. However, every node of a cluster hierarchy represents a set of vertices and faces rather than a single vertex[1]. At runtime, we maintain an *active cluster list* (ACL), which is similar to an active front in a vertex hierarchy and perform selective refinement on this list via the following operations:

- **Cluster-split:** A cluster in the active cluster list is replaced by its children.
- **Cluster-collapse:** Sibling clusters are replaced by their parent.

These operations are analogous to the vertex split and collapse in a vertex hierarchy but provide a more coarse-grained approach to selective refinement.

[1]In Chapter 3, we also used a cluster hierarchy, each cluster of which consists of portions of a vertex hierarchy. On the other hand, the cluster hierarchy for the CHPM representation consists of original or simplified geometry.

Progressive Meshes and Refinement

Each cluster contains a PM, which is a mesh sequence built from an input mesh by a sequence of edge collapse operations. The inverse operation, a vertex split, restores the original vertices and replaces the removed triangles. Each PM is stored as the most simplified or *base mesh* combined with a series of vertex split operations. In practice, refining a PM is a very fast operation and requires no dependency checks.

We use the notation M_A^0 to represent a base mesh of a cluster A. Moreover, M_A^i is computed by applying a vertex split operation to M_A^{i-1}. A PM can be refined within a range of object space error values. We refer to this range as the *error-range* of a cluster and is expressed as a pair: (*min-error*, *max-error*). The *max-error* is the error value associated with the base mesh (M^0) and the *min-error* is the error value associated with the highest resolution mesh (e.g. M_C^k, M_A^i and M_B^j as shown in Fig. 4.3).

The PMs allow us to perform smooth LOD transitions at the level of a single cluster. In order to perform globally smooth LOD transitions we require that the changes to the ACL between successive frames are also smooth. If cluster C is the parent of clusters A and B, we set the highest resolution mesh approximation of cluster C's PM to be the union of the base meshes of cluster A and B's PMs. That is, $M_C^k = M_A^0 \bigcup M_B^0$ (see Fig. 4.3). Therefore, the cluster-collapse and cluster-split operations introduce no popping artifacts.

Dual Hierarchies

The CHPM representation can be seen as dual hierarchies: an LOD hierarchy for view-dependent rendering and a bounding volume hierarchy (BVH) for occlusion culling. As an LOD hierarchy each interior cluster contains a coarser representation of its children's meshes. As a bonding volume hierarchy each cluster has an associated bounding volume (BV), which contains all the mesh primitives represented by its subtree. We use the

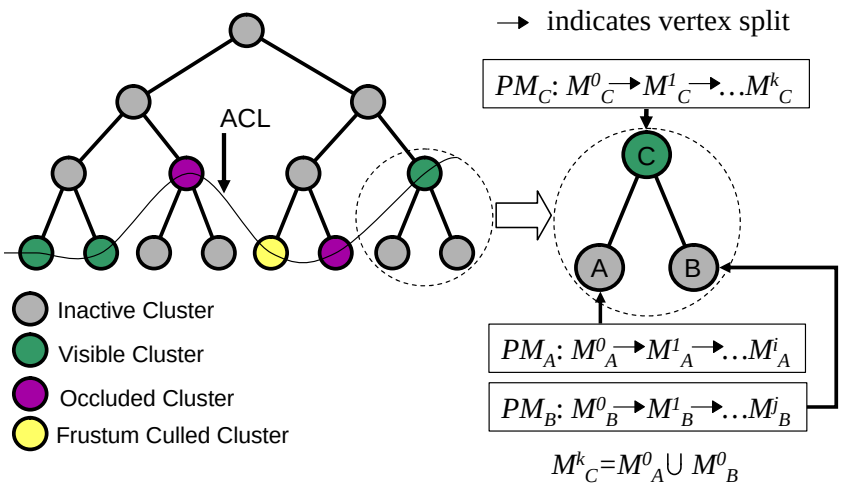


Figure 4.3: **Clustered Hierarchy of Progressive Meshes (CHPM):** At runtime the active cluster list (ACL) represents a front in the cluster hierarchy containing the clusters of the current mesh (left). Clusters on the ACL are classified as visible, frustum-culled, or occlusion-culled. The PMs (right) of visible clusters are refined to meet the screen space error bound by selecting a mesh from the PM mesh sequence. When the ACL changes, smooth LOD transitions occur because the most refined mesh of each PM is equal to the union of the base meshes of its children.

oriented bounding box as the BV representation.

By combining an LOD hierarchy with a BVH, we are able to improve a memory requirement of the representation and simplify the integration of view-dependent rendering and occlusion culling.

4.1.2 Algorithms

Quick-VDR consists of two major parts: a preprocess and a runtime rendering algorithm.

Preprocess: Given a large dataset, we compute a CHPM representation. Our out-of-core algorithm begins by decomposing the input mesh into a set of clusters. The clusters are passed to a cluster hierarchy generation algorithm which builds a balanced hierarchy in a top-down manner. We perform out-of-core hierarchical simplification

using the cluster hierarchy as a final step. We introduce *cluster dependencies* that allow boundary simplification while maintaining crack-free simplification and achieving efficient rendering performance at runtime.

Rendering Algorithm Quick-VDR uses the CHPM as a scene representation for out-of-core view-dependent rendering and occlusion culling. The CHPM is refined by performing two levels of refinement: a coarse-grained refinement at the cluster level and a fine-grained local refinement using a PM. Cluster dependencies assure that consistent cluster boundaries are rendered and that we are able to compute drastic simplifications. We use temporal coherence to accelerate refinement and to perform occlusion culling at the cluster level using hardware accelerated visibility queries. Quick-VDR uses the operating system's virtual memory manager through a memory mapped file for out-of-core rendering. In order to overcome the problem of accurately predicting the occlusion events, we introduce one frame of latency in the runtime pipeline. This allows us to load newly visible clusters to avoid stalling the rendering pipeline.

4.2 Building a CHPM

In this section we present an out-of-core algorithm to compute CHPMs for large datasets, such as CAD models, large isosurfaces, or scanned models. Our algorithm proceeds in three steps. First, we decompose the input mesh into a set of clusters. The decomposition occurs in several passes to avoid loading the entire input mesh at once. These clusters facilitate out-of-core access to the mesh for the remaining steps. Next, we construct the cluster hierarchy by repeatedly subdividing the mesh in a top-down manner. Finally, we compute progressive meshes for each cluster by performing a bottom-up traversal of the hierarchy.

4.2.1 Cluster Decomposition

The clusters form the underlying representation for both the preprocessing step as well as out-of-core view-dependent rendering with occlusion culling. We decompose the model into clusters, which are spatially localized portions of the input mesh. The generated clusters should be nearly equally sized in terms of number of triangles for several reasons. This property is desirable for out-of-core mesh processing to minimize the memory requirements. If the cluster size as well as the number of clusters required in memory at one time are bounded, then simplification and hierarchy construction can be performed with a constant memory footprint. Moreover, enforcing spatial locality and uniform size provides higher performance for occlusion culling and selective refinement.

The out-of-core cluster decomposition algorithm proceeds in four passes. The first three passes only consider the vertices of the original model and create the clusters while the fourth pass assigns the faces to the clusters. We use a variation of the cluster decomposition algorithm for out-of-core compression of large datasets presented in (Isenburg & Gumhold, 2003). However, our goal is to decompose the mesh for out-of-core processing and view-dependent rendering. As a result, we need only compute and store the connectivity information used by the simplification algorithm. To support transparent accesses on a large mesh during simplification, we also preserve all inter-cluster connectivity information.

Connectivity

It is desirable to have compact connectivity and easy access to the connectivity of out-of-core meshes during simplification. To meet these goals, we use corner-based connectivity for out-of-core meshes. A triangle consists of 3 corners, each of which has a index to an incident vertex and an index for the next corner that shares the same

incident vertex [2]. Each vertex also has an index, which indicates corner sharing the vertex. Since the vertices and triangles are grouped into clusters, we represent each index as two components: a cluster id and a local id. This index information can be packed in 4 bytes integer.

Given this connectivity information, we are able to support all the necessary operations (e.g. decimation operations) during simplification. Moreover, we can easily reconstruct the connectivity as we read triangles from the disk without storing them explicitly in the main memory.

Algorithm

The out-of-core cluster decomposition algorithm proceeds in four passes. The four passes of the algorithm are:

First vertex pass: We compute the bounding box of the mesh.

Second vertex pass: We compute balanced-size clusters of vertices (e.g. 3K vertices). Vertices are assigned to cells of a uniform 3D grid which may be subdivided to deal with irregular distribution of geometry. A graph is built with nodes representing the non-empty cells weighted by vertex count. Edges are inserted between each cell and its k nearest neighbors using an approximate nearest neighbor algorithm (Arya & Mount, 1993) (e.g. k=6). We use a graph partitioning algorithm (Hendrickson & Leland, 1995) to partition the graph and compute balanced-size clusters.

Third vertex pass: Based on the output of the partitioning, we assign vertices to clusters and reindex the vertices. The new index is a cluster/vertex pair that is used to locate the vertex in the decomposition. A mapping is created that maps the original vertex indices to the new pair of indices. This mapping can be quite large so it is stored

[2]For manifold meshes, we can use more compact *corner-table* proposed by Rossignac et al. (Rossignac et al., 2001).

in a file that can be accessed in blocks with LRU paging to allow the remainder of the preprocess to operate in a constant memory size.

Face pass: In the final pass, we assign each face to a single cluster that contains at least one of its vertices. The mapping file created in the previous pass is used to locate the vertices. The vertices of faces spanning multiple clusters are marked as constrained for simplification. These vertices make up the boundaries between clusters and are referred to as *shared vertices* while the remaining vertices are referred to as *interior vertices*.

The resulting cluster decomposition consists of manageable mesh pieces that can be transparently accessed in an out-of-core manner for hierarchy generation and simplification, while preserving all the original connectivity information. Different clusters computed from the dragon model are shown in Fig. 4.4.

4.2.2 Cluster Hierarchy Generation

In this section, we present an algorithm to compute the cluster hierarchy. The clusters computed by the decomposition algorithm described in the previous section are used as the input to hierarchy generation. Our goal is to compute a hierarchy of clusters with the following properties:

Nearly equal cluster size As previously discussed, consistent cluster size is important for memory management, occlusion culling, and selective refinement. Clusters at all levels of the hierarchy must possess this property.

Balanced cluster hierarchy During hierarchical simplification, cluster geometry is repeatedly simplified and merged in a bottom up traversal. The hierarchy must be well balanced so that merged clusters have nearly identical *error-ranges*.

Minimize shared vertices The number of shared vertices at the cluster boundary should be minimized for simplification. Otherwise, in order to maintain consistent

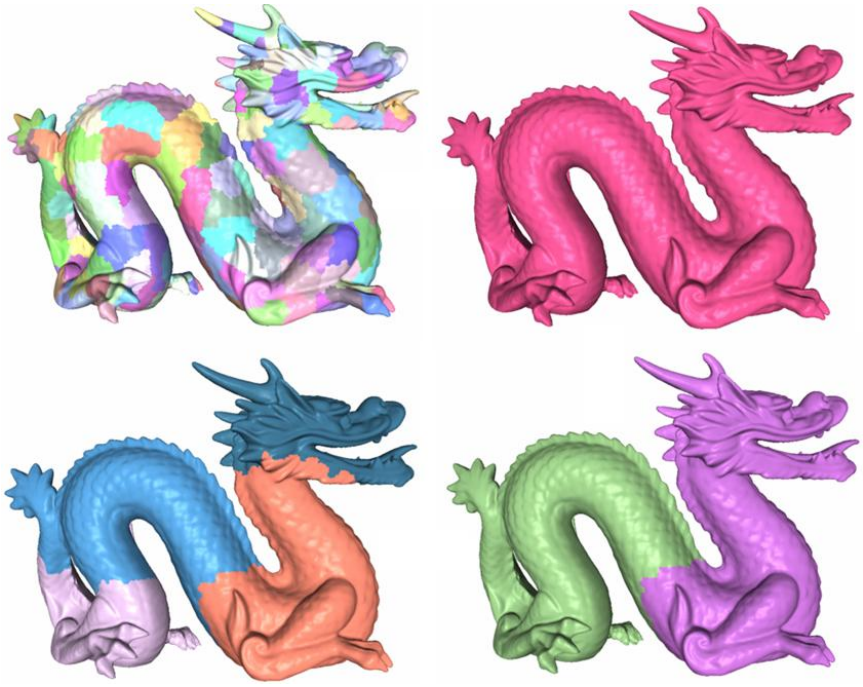


Figure 4.4: **An Example of Cluster Hierarchy:** These images highlight different clusters of the hierarchy of the dragon model. The leaf clusters are shown in the top left image. The root cluster is shown in the top right, the second level clusters are shown in the bottom right, and the third level clusters are shown in the bottom left.

cluster boundaries, the simplification will be over-constrained and may result in lower fidelity approximations of the original model.

High spatial locality The cluster hierarchy should have high spatial locality for occlusion culling and selective refinement.

We achieve these goals by transforming the problem of computing a cluster hierarchy into a graph partitioning problem and compute the hierarchy in a top down manner. Each cluster is represented as a node in a graph, weighted by the number of vertices. Clusters are connected by an edge in the graph if they share vertices or are within a threshold distance of each other. The edges are weighted by the number of shared vertices and the inverse of the distance between the clusters, with greater priority placed on the number of shared vertices. The cluster hierarchy is then constructed in a top-

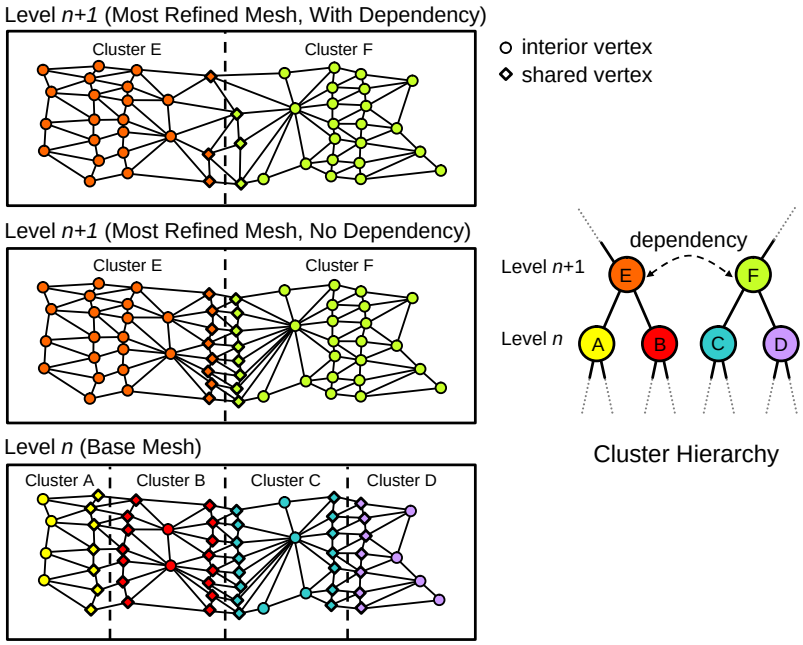


Figure 4.5: **Dependencies:** After simplifying level n of the hierarchy the boundaries AB, BC, and CD are all under-simplified because they are constrained. When initializing the base meshes of E and F prior to simplifying level $n+1$, two of these boundaries, AB and CD, are no longer constrained because they have been merged. The boundary BC was not merged and will remain under-simplified. We can, however, simplify the faces along this boundary if we mark E and F as dependent.

down manner by recursively partitioning the graph into halves considering the weights, thus producing a binary tree. The weights guide the partitioning algorithm (Karypis & Kumar, 1998) to produce clusters with spatial locality while tending towards fewer shared vertices. The top down partitioning creates an almost balanced hierarchy. An example of the cluster hierarchy of the dragon model is shown in Fig. 4.4.

4.2.3 Out-of-Core Hierarchical Simplification

We simplify the mesh by traversing the cluster hierarchy in a bottom-up manner. Each level of the cluster hierarchy is simplified in a single pass so the simplification requires $\lceil log_2(n) + 1 \rceil$ passes where n is the number of leaf clusters. During each pass only the

cluster being simplified and clusters with which it shares vertices must be resident in memory.

Simplification operations are ordered by a priority queue based upon quadric errors (Garland & Heckbert, 1997). We build the progressive meshes (PMs) for each cluster by applying "half-edge collapses". The half-edge collapse, in which an edge is contracted to one of the original vertices, is used to avoid creation of new vertices during simplification. Edges adjacent to shared vertices are not collapsed during simplification. The edge collapses and associated error values are stored along with the most refined mesh of a PM. After creating the PM, the *error-range* of the cluster is computed based on the errors of the PM's original and base mesh.

When proceeding to the next level up the hierarchy, the mesh within each cluster's PM is initialized by merging the base meshes of the children. Constraints on vertices that are shared by two clusters being merged are removed thereby allowing simplification of the merged boundary. Since the intermediate clusters should be nearly the same size as the leaf level clusters, each cluster is simplified to half its original face count at each level of the hierarchy.

As simplification proceeds a file is created for the progressive mesh of each cluster. However, handling many small files is inefficient at runtime. The PM files are merged into one file which can be memory mapped to allow the OS to perform memory management of the PMs and optimize disk access patterns during runtime rendering. The file is stored in a breadth first manner in an attempt to match the probable access pattern during runtime refinement.

4.2.4 Boundary Constraints and Cluster Dependencies

In order to support out-of-core rendering and to allow efficient refinement at runtime, it should be possible to refine the PM of each cluster independently and at the same

time maintain a crack-free consistent mesh. To achieve this, our algorithm detects the shared vertices and restricts collapsing the edges adjacent to them during hierarchical simplification. As simplification proceeds up the hierarchy, these constraints are removed when the clusters sharing the vertices have been merged.

While these constraints assure crack-free boundaries between clusters at runtime, they can be overly restrictive. After simplifying several levels of the hierarchy most of the vertices in the base mesh of the PM are shared vertices. As illustrated in Fig. 4.5 this problem arises along boundaries between clusters that are merged at higher levels in the hierarchy. This can degrade the quality of simplification, and impedes drastic simplification. In Fig. 4.5 notice that the boundaries between clusters A and B and clusters C and D are merged in the next level of the hierarchy (E and F). However, the boundary between B and C is not merged until higher up the hierarchy, but is already drastically under-simplified compared to the interior. This constraint problem is common to many hierarchical simplification algorithms that decompose a large mesh for view-dependent rendering (Hoppe, 1998; Prince, 2000) or compute hierarchies of static LODs (HLODs) (Govindaraju et al., 2003c).

We introduce *cluster-level dependencies* to address this constraint problem. The intuition behind dependencies is that precomputed simplification constraints on shared vertices can be replaced by runtime dependencies. During hierarchical simplification, we may collapse an edge adjacent to a shared vertex. The clusters sharing that vertex are marked as dependent upon each other. Boundary simplification occurs on the merged meshes prior to PM generation thereby allowing the computed PMs to be refined independently at runtime. In Fig. 4.5 clusters E and F are marked dependent and thereby allow the boundary to be simplified.

At runtime, splitting a cluster forces all its dependent clusters to split so that the boundaries are rendered without cracks. Likewise, a parent cluster cannot be collapsed

unless all of its dependent clusters have also been collapsed. In Fig. 4.5, clusters E and F must be split together and clusters A, B, C, and D must be collapsed together (assuming E and F are dependent). For example, if clusters B and F are rendered during the same frame, their boundary will be rendered inconsistently and may have cracks.

Dependencies Criteria Although cluster dependencies allow boundary simplification, we need to use them carefully. Since splitting a cluster forces its dependent clusters to split, dependencies will cause some clusters to be rendered that are overly conservative in terms of view-dependent error bounds. Furthermore, the boundaries change in one frame which may cause popping artifacts. This can be exacerbated by "chained" dependencies in which one cluster is dependent upon another cluster which is in turn dependent upon a third cluster, and so on.

To avoid these potential runtime problems, we prioritize clusters for boundary simplification. At each level of hierarchical simplification the clusters are entered into a priority queue. Priorities are assigned as the ratio of average error of shared vertices to the average error of interior vertices. A cluster, A, is removed from the head of the priority queue. For each cluster, B, that shares at least j (e.g. 5) vertices with A we apply boundary simplification between A and B if the following conditions are met:

1. A and B will not be merged within a small number of levels up the cluster hierarchy (e.g., 2).

2. A and B have similar *error-ranges*.

3. A dependency between A and B will not introduce a chain (unless all the clusters in the chain share vertices).

This is repeated for each cluster in the priority queue. The first condition avoids creating dependencies between clusters that are resolved within only a few additional

hierarchy levels. The second condition discourages dependencies between those clusters that are unlikely to be simultaneously present in the ACL at runtime. The third condition prevents long dependency chains and preserves selective refinement at the cluster level. The cluster dependencies ensure that a sufficient number of shared vertices are collapsed at each level of the hierarchy while still generating and rendering crack-free simplifications at runtime. An example of posing cluster dependencies in Lucy model is shown in Fig. 4.6.

4.2.5 Buffer-based Processing

The hierarchical simplification algorithm described in the previous section carefully computes dependencies between the clusters for high quality simplification and fast rendering. However, this algorithm would access the clusters in a random order due to the priority queue representation. This can degrade the performance of the simplification preprocess. To overcome this problem, we apply *buffer-based processing* to improve the access pattern of the algorithm. The buffer is used to hold the clusters, which are input to the priority queue. The size of a buffer is limited by maximum memory used for preprocessing. As we traverse the clusters in a breadth first order, we allocate visited-clusters into the buffer and compute their priorities in the priority queue. Once the visited-clusters reach the maximum size of the buffer, the simplifications are processed according to the priorities within the priority queue. The priority queue is then emptied and processing is continued until all the clusters have been visited by the traversal algorithm.

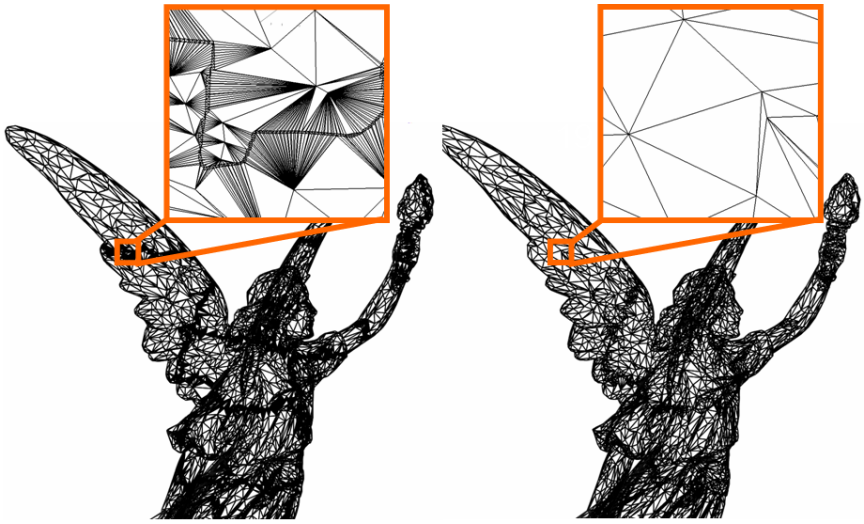


Figure 4.6: **An Example of Cluster Dependencies:** These images highlight meshes with and without posing cluster dependencies in the Lucy model. The left image shows a simplified Lucy model without the cluster dependencies. It consists of 227K triangles at the specified (20) pixels of error in 512 by 512 image resolution. The right images shows a simplified Lucy model with cluster dependencies. It is composed of only 19K triangles with the same pixels of error. Therefore, we are able to achieve more than 1 order of magnitude speedup on rendering time by using cluster dependencies. The zoomed areas are shown in oranges rectangles.

4.3 Interactive Out-of-Core Display

In the previous section, we described an algorithm to compute CHPM. In this section, we present a novel rendering algorithm that uses CHPM representation for occlusion culling, view-dependent refinement and out-of-core rendering. The entire representation including the PMs is stored on the disk. We load the coarse-grained cluster hierarchy into main memory and keep a working set of PMs in main memory. The cluster hierarchy without the PMs is typically a few megabytes for our benchmark models (e.g. 5MB for St. Matthew model). We perform coarse-grained refinement at the cluster level and fine-grained refinement at the level of PMs. We introduce a frame of latency in the rendering pipeline in order to fetch the PMs of newly visible clusters from the disk and avoid stalls in the rendering pipeline.

4.3.1 Simplification Error Bounds

A key issue of view-dependent refinement is computation of errors associated with the LODs generated at runtime. The allowable runtime error is expressed in screen-space as a *pixels-of-error* (POE) value. Using the POE value and the minimum distance between a cluster and the viewpoint, we compute the maximum object-space error that is allowed for the cluster. We call this value the *view-dep-error-bound*. We use the *view-dep-error-bound* for a cluster to refine both clusters and PMs. This approach allows us to efficiently perform view-dependent computations using a single object-space comparison between the *view-dep-error-bound* and a stored error value in the clusters and PMs.

4.3.2 View-Dependent Refinement

View-dependent refinement of the CHPM representation is similar to the refinement operations explained in Chapter 3.3.1. Our algorithm maintains an active cluster list (ACL), which is a cut of clusters in the hierarchy representing the scene. During each frame, we refine the ACL based on the current viewing parameters. Specifically, we traverse the ACL and compute the *view-dep-error-bound* for each cluster. Each cluster on the active front whose *view-dep-error-bound* is less than the *min-error* of its *error-range* is split because the PM cannot meet the *view-dep-error-bound*. Similarly, sibling clusters that have a greater *view-dep-error-bound* than *max-error* are collapsed. Each PM in the ACL is refined prior to being rendered by choosing the mesh in the PM mesh sequence with the lowest face count that meets the *view-dep-error-bound*.

To accelerate the view-dependent refinement, we take advantage of temporal coherence between successive frames. We start with the position within the PM from the previous frame and perform the aforementioned view-dependent computation.

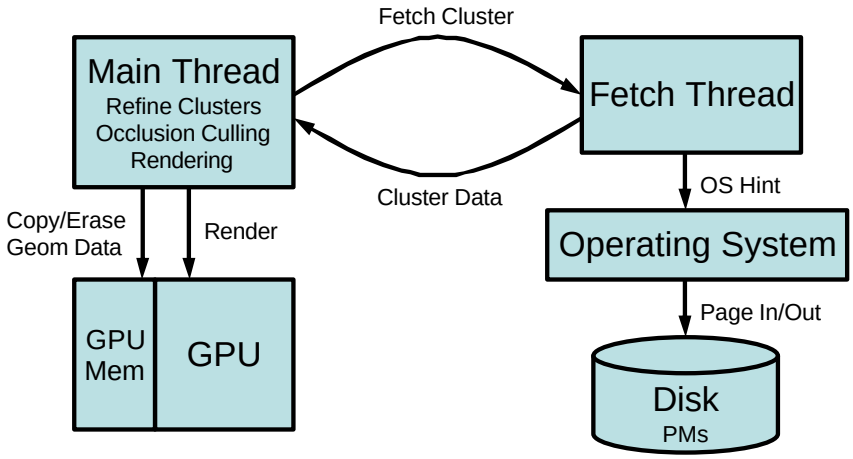


Figure 4.7: **Overall Data Flow:** Quick-VDR uses two threads. The fetch thread manages the out-of-core PMs through interaction with the operating system. The main thread handles refinement, occlusion culling, and rendering. Geometry data for a working set of PMs is stored in GPU memory.

4.3.3 Handling Cluster Dependencies

Our simplification algorithm introduces dependencies between the clusters so that we can simplify their boundaries during the preprocess. We use these dependencies to generate a crack-free simplification at runtime.

Handling cluster dependencies is very similar to an approach of handling vertex dependencies. Cluster-collapses occur to reduce the polygon count in the current refinement. However, prior to collapsing a pair of sibling clusters we must check the parent's dependencies. If the children of any dependent clusters cannot also be collapsed, then the initial cluster collapse cannot occur. Cluster-splits also occur to increase the polygon count to meet the error bound. If there are dependencies on a parent cluster, we force the cluster-splits of all the dependent clusters as well as the parent cluster. These checks occur at the cluster level and are inexpensive.

4.3.4 Conservative Occlusion Culling

Our occlusion culling algorithm is based on the previous culling algorithm explained in Chapter 3.3.3. We simplify the previous culling algorithm due to the simplicity of the CHPM representation.

We exploit temporal coherence in occlusion culling. Each frame our algorithm computes a potentially visible set of clusters (PVS) and a newly visible set (NVS), which is a subset of the PVS. The PVS for frame i is denoted as PVS_i and the NVS as NVS_i. An occlusion representation (OR_i), represented as a depth buffer, is computed by rendering PVS_{i-1} as an occluder set. Using OR_i we determine PVS_i. The overall rendering algorithm is:

Step 1: Refine ACL. The ACL is refined as described in Sec. 4.3.2 based on the camera parameters for frame i.

Step 2: Render PVS_{i-1} to compute OR_i: We refine clusters in PVS_{i-1} based on the viewpoint, compute a simplification for each cluster and render them to compute OR_i. OR_i is represented as a depth map that is used for occlusion culling. These clusters are rendered to both the depth and color buffers.

Step 3: Compute NVS_i and PVS_i: The bounding boxes of all the clusters in the ACL are tested for occlusion against OR_i. This test is performed with hardware occlusion queries at the resolution of image precision. The depth and color writes are disabled during this step to prevent overwriting of the depth and color values from Step 2. PVS_i contains all the clusters with visible bounding boxes, while NVS_i contains the clusters with visible bounding boxes that were not in PVS_{i-1}.

Step 4: Render NVS_i: The PMs of clusters in NVS_i are refined and rendered, generating the final image for frame i.

4.3.5 Out-of-Core Rendering

Our algorithm works with a fixed memory footprint of main memory and graphics card memory. The entire cluster hierarchy is in main memory and we fetch the PMs of the clusters needed for the current frame as well as prefetch some PMs of clusters for subsequent frames. Additionally, we store the vertices and faces of active clusters in GPU memory. By rendering the primitives directly from GPU memory, AGP bus bandwidth requirement is reduced and we obtain an increased triangle throughput.

Out-of-core Framework

Our out-of-core rendering algorithm uses the paging mechanism in the operating system by mapping a file into read-only logical address space (Lindstrom & Pascucci, 2002). We choose the OS's virtual memory management because it can effectively optimize the disk access patterns and perform efficient memory management, which simplifies the design of our out-of-core algorithm. However, an application controlled paging mechanism can improve the performance of out-of-core memory management (Cox & Ellsworth, 1997). To fully take advantage of this mechanism, we store our view-dependent representation in a memory coherent manner, as described in Sec 4.2.3. However, there is a limitation (e.g. 2GB in Windows XP) in 32bit machine for mapping a file to user-accessible address space. We overcome this limitation by mapping only a 32MB portion of the file at a time and remapping when data is required from outside this range.

Our out-of-core rendering algorithm uses two separate threads: a main thread and a fetch thread. The rendering thread performs view-dependent refinement, occlusion culling and rendering. The fetch thread is used to prepare data for PMs that are likely to be used in the future. This thread provides hints to OS and converts the PM data to the runtime format. The overall data flow is shown in Fig. 4.7.

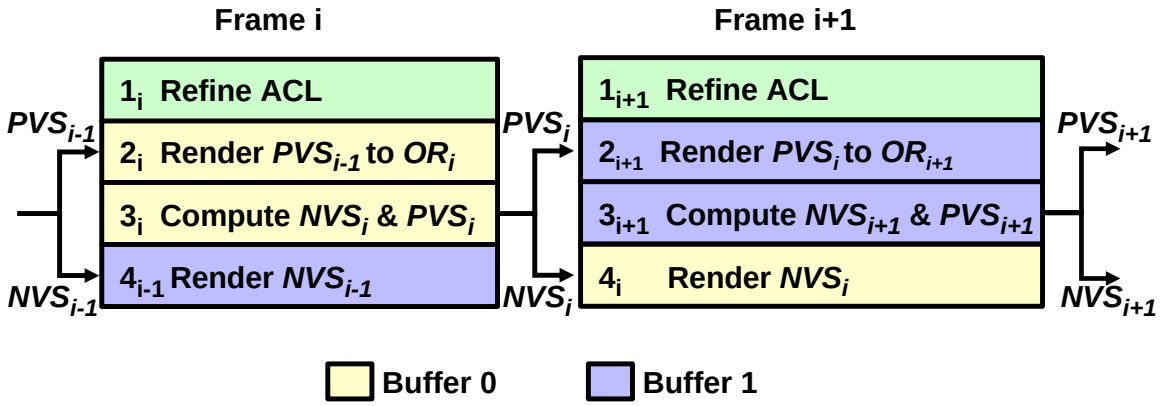


Figure 4.8: **Our Rendering Pipeline:** In frame i occlusion culling is performed for frame i but the final image for frame $i-1$ is displayed. This allows extra time for loading the PMs of newly visible clusters. Two off-screen buffers facilitate this interleaving of successive frames. The partial rendering of frame i is stored in one buffer while occlusion culling for frame $i+1$ occurs in the other buffer.

LOD Prefetching

When we update clusters in the ACL by performing cluster-collapse and cluster-split operations, the children and parent clusters are activated. The PMs of these clusters may not be already loaded in the main memory and this can stall the rendering pipeline. To prevent these stalls whenever a cluster is added to the ACL we prefetch its parent and children clusters. Thus, we attempt to keep one level of the hierarchy above and below the current ACL in memory.

Visibility Fetching

Predicting visibility or occlusion events is difficult, especially in complex models with high depth complexity and small holes. As a result, our algorithm introduces a frame of latency in the rendering pipeline and fetches the PMs of the newly visible cluster in the ACL from the disk.

In our rendering algorithm visibility events are detected in Step 3, and the newly

visible clusters are added to NVS_i (Sec. 4.3.4). These clusters are then rendered in Step 4, which will likely not allow enough time to load these clusters without stalling. Step 2, rendering OR_i, is the most time consuming step of the rendering algorithm. Therefore, we delay the rendering of NVS_i until the end of Step 2 of the next frame and render PVS_{i-1} while fetching PMs from the disk in parallel (as shown in Fig. 4.7). Our rendering pipeline is reordered to include a frame of latency thereby increasing the time allowed to load a cluster to avoid stall.

During frame i we perform Steps 1 through 3 of the rendering algorithm with the camera parameters for frame i. However, we perform Step 4 for frame $i-1$ and generate the final image for frame $i-1$. The overall pipeline of the algorithm proceeds as: 1_i, 2_i, 3_i, 4_{i-1}, 1_{i+1}, 2_{i+1}, 3_{i+1}, 4_i, ..., where n_j refers to Step n of frame j (as shown in Fig. 4.8).

In this reordered pipeline, the PM of a cluster in NVS_i will first have to be rendered during Step 2_{i+1} as this step renders PVS_i and clusters added to NVS_i are also added PVS_i in Step 3_i (refer to Fig. 4.8). During Step 2_{i+1} we first render all the PMs that are already in memory. Since this is the most time consuming step of the rendering algorithm, most of the PMs of the newly visible clusters are loaded during this time. As a result, we are able to balance the load between fetching PMs from the disk and rendering without stalls.

To implement this pipeline, we use a pair of off-screen buffers. One buffer holds the partial rendering of a frame from Step 2 so that it may be composited with the newly visible clusters in Step 4 the following frame. The odd numbered frames use the first buffer while the even-numbered frames use the second buffer, so that each consecutive pair of frames can render to separate buffers. Fig. 4.8 illustrates how the buffers are used for two consecutive frames.

4.3.6 Utilizing GPUs

We achieve high throughput from graphics cards by storing the mesh data on the GPU, thereby reducing the data transferred from the CPU to the GPU during each frame. We use the GL_ARB_vertex_buffer_object OpenGL extension that performs GPU memory management for both the vertex and the face arrays. We use the half-edge collapse decimation operation so that the set of vertices used in the PMs is a subset of the vertices of the original model. However, we generate some new faces during each frame by performing vertex splits or edge collapse operations during local refinement of each PM. In practice, only a small number (e.g., 5%) of PMs require refinement during each frame. As a result, we only transmit the faces of these PMs to the GPU and the other faces are cached in the GPU memory. By utilizing the GL_ARB_vertex_buffer_object OpenGL extension, we have been able to achieve an average throughput of 23 million triangles on a PC with GeForce 5950FX Ultra GPU.

4.4 Implementation and Performance

In this section we describe our implementation and highlight its performance on massive models.

4.4.1 Implementation

We have implemented our out-of-core simplification and runtime system on a dual 2.4GHz Pentium-IV PC, with 1GB of RAM and a GeForce 5950FX Ultra GPU with 128MB of video memory. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files.

We use the METIS graph partitioning library (Karypis & Kumar, 1998) for cluster computation. Since the METIS library does not guarantee that partitioned graphs are

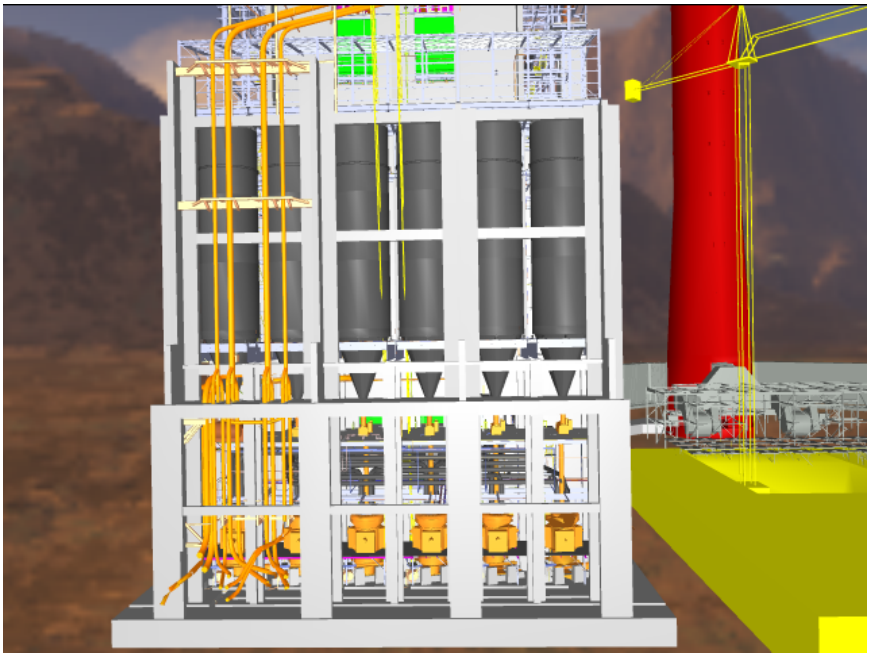


Figure 4.9: **Power Plant rendered by Quick-VDR:** A rendering of the power plant model using our runtime algorithm. This model consists of over 12M triangles and has high depth complexity with small occluders. It is rendered at an average of 28 FPS using 400MB of main memory by our system.

connected, we perform a post-processing step to connect partitioned graphs if possible. We use NVIDIA OpenGL extension GL_NV_occlusion_query to perform occlusion queries. We are able to perform an average of approximately 400K occlusion queries per second on the bounding boxes. In practice, the ACL consists of hundreds of clusters, and we are able to perform occlusion culling in $1-3$ milliseconds per frame.

4.4.2 Massive Models

Our algorithm has been applied to three complex models, a coal-fired power plant composed of more than 12 million polygons and 1200 objects (Fig. 4.9), the St. Matthew model consisting of a single 372 million polygon object (Fig. 4.2), and an isosurface model consisting of 100 million polygons (Fig. 1). The details of these models are shown in Table 4.1. We generated paths in each of our test models and used them to

Model	PP	Isosurface	St. Matthew
Triangles (M)	12.2	100	372
Original Size (MB)	485	2,543	9,611
Num Clusters (K)	5.8	16	65
Memory footprint used (MB)	32	256	512
Size of CHPM (MB)	625	3,726	13,992
Size of cluster hierarchy (MB)	0.3	2.1	5.1
Processing time (min)	35	182	682

Table 4.1: **Preprocess of Quick-VDR:** *Preprocess timings and storage requirements for test models. We are able to compute a CHPM for each environment using our out-of-core algorithm and a memory footprint of* $256 - 512MB$.

test the performance of our algorithm.

4.4.3 Performance

We have applied our out-of-core CHPM generation preprocess to each of the models. Table 4.1 presents preprocessing time for each model on the PC. We guarantee the maximum memory requirement is less than a user specified threshold (e.g. 256MB). We fully utilize the specified memory footprint during the cluster decomposition. However, other preprocessing steps including the cluster hierarchy generation and hierarchical simplification require very small portion of the memory footprint. Hierarchical simplification takes approximately 85% of the preprocess time. The remainder of the time is dominated by the face pass of the cluster decomposition. This pass makes random accesses to the out-of-core vertex index mapping table to locate face vertices in the cluster decomposition. We could use an external sort of the mapping table to improve access patterns as in (Lindstrom & Silva, 2001).

We are able to render all these models at interactive rates (10-35 frames per second) on a single PC. In Table 4.2 we report the runtime performance of our algorithm on the three models.

Model	POE	Avg FPS	Avg Front # Clusters	Avg # Ecol/Vsplit	Avg # Tri(K)
Power plant	1	26	2279	181	592
Isosurface	20	24	372	488	920
St. Matthew	1	17	434	2196	1121

Table 4.2: **Runtime Performance:** We highlight the performance on the three benchmarks. The average frame rate, average front size, and average number of edge collapse and vertex splits per frame are presented for a sample path in each model. All the data is acquired at 512×512 resolution. We use a 400MB memory footprint for the power plant model and 600MB for other models.

Fig. 4.10 illustrates the performance of the system on a complex path in isosurface model. Table 4.2 shows the average frame rate, front size, and number of edge collapse and vertex split operations performed for paths in each of our test models.

Table 4.3 shows the average breakdown of the frame time for each model. Rendering costs dominate the frame time.

Out-of-core

Our system relies on the underlying operating systems virtual memory management for paging of PMs and, as discussed in Sec. 4.3.5, uses a frame of latency to hide load times of newly visible clusters. The frame rates of a sample path of the isosurface model are shown in Fig. 4.10. Please note that there is no severe stalling that would cause large downward spikes in the frame rate. We achieve an average frame rate of 24 frames per second.

Occlusion culling

Occlusion culling is very important for rendering models with high depth complexity such as the power plant and isosurface models. Fig. 4.10 highlights the benefit of occlusion culling by comparing the frame rate of our system over a path with occlusion

culling enabled and disabled. On average the frame rate is $25 - 55\%$ higher when occlusion culling enabled. However, we achieve the improvement by spending very small portion (e.g., less than 3%) of the frame time on occlusion culling (see Table 4.3). This suggests that we can further improve the overall runtime performance by reducing the granularity of occlusion culling. One possible choice is to use *sub-objects* in each cluster for occlusion culling.

4.5 Analysis and Limitations

In this section, we analyze the performance of Quick-VDR. We also highlight the benefits over prior approaches and describe some of its limitations.

Refinement Cost of CHPMs vs. Vertex Hierarchies: Most of the earlier algorithms for view-dependent simplification use a vertex hierarchy. These algorithm compute an active vertex front in the hierarchy and handle dependencies at the vertex or edge level.

We compared the refinement cost of CHPM with an implementation of a vertex hierarchy (VDPM) for an isosurface with about 1M triangles (see Table 4.4). We have observed that CHPM refinement cost is one-two orders of magnitude lower, even without occlusion culling. This lowered cost is due to the following factors:

1. Our clusters consist of thousands of triangles. As a result, the size of ACL is typically more than one-two orders of magnitude smaller than the size of active front in a vertex hierarchy.

2. We perform coarse-grained refinement operations at the cluster level and use a single conservative error bound for an entire cluster. Therefore, refinement of

individual PMs is much faster than it would be by performing per-vertex computations across an active vertex front.

3. Handling dependencies at the cluster level is significantly cheaper than those at the vertex level.

Furthermore, occlusion culling helps us in further reducing the refinement cost as we do not need to refine PMs of the clusters that are not visible.

Conservative Occlusion Culling: Quick-VDR performs conservative occlusion culling up to image precision. The occlusion computations are performed at the cluster level. The size of ACL is typically a few hundred clusters so performing occlusion culling takes $1 - 10\%$ of the total frame time.

Storage Overhead: Our CHPM implementation requires on average 88MB per million vertices. This is low compared to Hoppe's (Hoppe, 1997) VDPM representation (224MB) and XFastMesh (108MB) (DeCoro & Pajarola, 2002). Moreover, CHPM can easily represent models with non-manifold topologies. According to Hoppe (Hoppe, 1998) the compression ratio of PMs decreases as the size of the model increases. In a CHPM the size of each PM is independent of the total mesh size. Furthermore, we can use the relationship between the PMs of a parent cluster with its children cluster to achieve higher compression.

Out-of-Core Computation: Our out-of-core preprocess is able to construct a CHPM from large datasets using a constant-sized memory footprint. Moreover, our hierarchical simplification algorithm produces nearly in-core quality progressive meshes.

Our current implementation does not achieve the same performance of Lindstrom (Lindstrom, 2003) in terms of triangles simplified per second. Lindstrom (Lindstrom, 2003) applies external-memory sorts to his out-of-core data structures to improve the

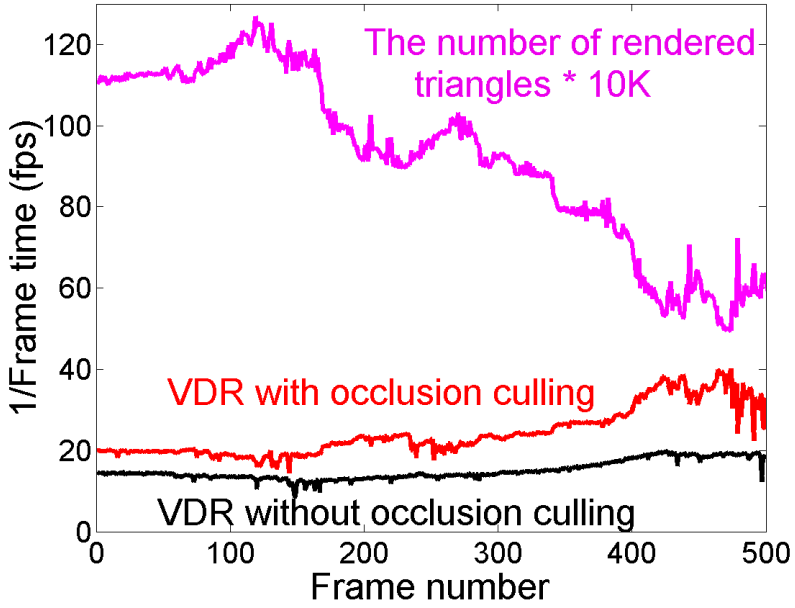


Figure 4.10: **Frame Rate in Isosurface Model:** Frame rates are shown for a sample path using our system. For comparison we show our system without occlusion culling. The number of rendered triangles are also shown.

access patterns and we can also use them to improve the performance of our system. However, Lindstrom (Lindstrom, 2003) does not preserve all the faces and vertices in the leaf level of the hierarchy. This disadvantage is recently addressed (Shaffer & Garland, 2005).

Quick-VDR introduces a frame of latency to fetch PMs of the newly visible cluster from the disk. This is needed to take into account the visibility events that can occur between successive frames. Earlier algorithms that combine visibility computations with out-of-core rendering decompose large CAD environments into rectangular cells and do not introduce additional latency (Aliaga et al., 1999; Correa et al., 2002). However, it may not be easy to decompose large isosurfaces for visibility-based prefetching. Moreover, the MMR system (Aliaga et al., 1999) uses image-based impostors and can introduce additional popping artifacts.

4.5.1 Comparisons

In this section, we compare the performance of our algorithm with prior approaches. Our algorithm has been applied to complex models composed of a few hundred million polygons. In contrast, view-dependent algorithms were applied to scanned models with $8-10$ million triangles (DeCoro & Pajarola, 2002), 2M triangle isosurface and the power plant model (See Chapter 3) or were combined with approximate occlusion culling (El-Sana & Bachmat, 2002). Lindstrom's algorithm (Lindstrom, 2003) does not perform occlusion culling and has been applied it to a 47M triangle isosurface. It is difficult to perform a direct comparisons with these approaches as they used an older generation of the hardware and it may not have the same set of features (e.g. occlusion queries).

The main reasons for the high frame-rate performance of Quick-VDR on massive models are:

- Low refinement cost during each frame.
- High GPU throughput obtained by rendering PMs directly from GPU memory
- Significant occlusion culling based on the cluster hierarchy.
- Out-of-core computations at the cluster level.

Adaptive TetraPuzzles

The Adaptive TetraPuzzles approach (Cignoni et al., 2004) has been proposed for rendering models consisting of hundreds of millions of triangles. Adaptive TetraPuzzles uses a precomputed regular hierarchy of tetrahedra to spatially partition the model. A set of tetrahedrons sharing a longest edge form a "diamond" and are subdivided together. Each tetrahedron contains static LODs, which are stored as indexed triangle strips for faster rendering. On the other hand, CHPM employs progressive meshes in

a cluster hierarchy to further provide smooth local refinement and reduce the number of triangles in each cluster. As a result, it is hard to compare the rendering quality of the images generated by these algorithms.

To simplify boundary triangles for high-quality simplification and faster rendering performance at runtime, we introduce explicit dependencies between arbitrary clusters. On the other hand, in a regular hierarchy of tetrahedra, boundary triangles alternate between different levels of the hierarchy based on diamonds. This can be also thought of posing *implicit cluster dependencies*. The TetraPuzzles approach has been applied to scanned models, which have uniform distribution of geometry. Therefore, it is not clear whether the regular hierarchy of tetrahedra with implicit dependencies works with a non-uniform distribution of geometry. Moreover, the effectiveness of occlusion culling with the tetrahedron hierarchy has not been evaluated.

Far Voxels

Recently, Gobbetti and Marton (Gobbetti & Marton, 2005) proposed *Far Voxels*, which is a hybrid multiresolution representation: each leaf node of a multiresolution hierarchy has a set of original triangles while each intermediate node has point clouds, which are volumetric simplified representations of triangles contained in its sub-trees. Intermediate nodes are only used when they are projected on less than one pixel in the image screen. By using volumetric representations for intermediate nodes, they are able to drastically simplify complex geometry, which cannot be easily achieved by polygon simplification. However, their method does not provide conservative error metrics that measure geometric errors introduced by the volumetric representations. Moreover, their method does not work well with various illumination models such as specular lighting.

Model	Refining	Occlusion Culling	Rendering	Stalling
Power plant	1.8%	13.9%	83.3%	1.0%
Isosurface	2.2%	6.6%	90.1%	1.1%
St. Matthew	4.1%	1.4%	93.8%	0.7%

Table 4.3: Runtime Timing Breakdown. *This graph shows the percentage of frame time spent on the four major computations of the runtime algorithm. More than* 80% *of the time is spent in rendering the potential occluders and visible primitives. The overhead of performing occlusion queries, refinement and stalling is relatively small.*

iWalk System

The iWalk system (Corrêa et al., 2003; Corrêa, 2004) has been proposed for interactive rendering of large models. iWalk can support high-resolution (4096×3072) and multi-tiled displays by employing a sort-first parallel out-of-core rendering. iWalk also use static LODs and an octree for conservative occlusion culling. At runtime, iWalk predicts visibility events based on visibility coefficients stored in the octree nodes. iWalk was applied to the power plant model and an isosurface model consisting of 473 million triangles. Because iWalk system has been tested in clusters of low-end commodity PCs, it is difficult to directly compare performance with Quick-VDR. iWalk achieves 10 frames per second on the power plant model and 8 frames per second on average for the isosurface model with 8 rendering servers.

4.5.2 Limitations

The main limitation of our approach is one frame of latency in the rendering pipeline. Other limitations include:

Drawbacks of CHPM: The set of possible dynamic simplifications represented by a CHPM is less than that of a full vertex hierarchy. This is caused by decomposing the model into clusters and representing each cluster as a linear sequence of edge collapses.

Method	Vertex Hierarchy	CHPM
Num. Dependency Checks	4.2M	223
Refinement Time(ms)	1,221	32

Table 4.4: **Refinement Performance of CHPM and VH:** A comparison of refinement cost between a CHPM and vertex hierarchy in a 1M triangle isosurface. This table measures the time to fully refine the mesh from the base mesh. The number of dependency checks for the vertex hierarchy is the sum of the number of triangles that are stored in the dependencies of the vertex nodes. The number of dependency check for the CHPM representation is similarly computed based on the count of clusters.

As a result a single *view-dep-error-bound* value must be used for an entire cluster. Thus, a given screen space error may be met with a slightly higher triangle count by a CHPM than a vertex hierarchy.

Dependencies: Cluster dependencies that force us to perform additional cluster-split operations might cause popping artifacts. These popping artifacts are always from lower to higher LOD which may be preferable to pops that decrease LOD.

Occlusion culling and coherence: Our occlusion culling algorithm assumes high temporal coherence between successive frames. Its effectiveness varies as a function of coherence between successive frames. Furthermore, if a scene has very little or no occlusion, the additional overhead of performing occlusion queries could lower the frame rate.

Cluster decomposition and hierarchy: The performance of our preprocessing and runtime algorithm depends on good cluster decomposition and hierarchy generation. We reduce the problem to graph partitioning algorithms and the current algorithms for partitioning cannot guarantee a good decomposition for every input.

Chapter 5

Approximate Collision Detection

Collision detection has been well-studied for more than three decades and some of the commonly used algorithms are based on spatial partitioning or bounding volume hierarchies (BVH). However, existing algorithms may not achieve interactive performance on large models consisting of tens of millions of polygons. The memory requirements of these algorithms are typically very high, as precomputed BVHs can take many gigabytes of space. Moreover, the number of pairwise overlap tests between the bounding volumes can grow as a super-linear function of the model size, thereby slowing down the query performance.

In order to deal with the model complexity, algorithms using multiresolution representations or model simplification techniques have been proposed. These algorithms have been used to generate tight fitting BVHs (Tan et al., 1999), to create static contact LODs (Otaduy & Lin, 2003), and to evaluate various factors affecting collision perception (O'Sullivan & Dingliana, 2001). To the best of our knowledge, none of them have been applied to general, unstructured complex models composed of millions of triangles.

In this chapter we propose a fast and conservative collision detection algorithm for massive models composed of tens of millions of polygons. We use a clustered hierarchy of progressive mesh (CHPM) as a model representation to provide approximate colli-

sion detection using dynamic simplification. The CHPM representation serves as a *dual hierarchy* of each model. We use this representation both as a bounding volume hierarchy to cull away cluster pairs that are not in close proximity and as a multiresolution representation that adaptively computes a simplified representation of each model on the fly. Our algorithm utilizes the cluster hierarchy for coarse-grained refinement and progressive meshes (PMs) associated with each cluster for fine-grained local refinement. This allows us to rapidly compute a dynamic simplification and reduce the "popping" or discontinuities between successive collision queries associated with static levels of detail (LODs). We use GPU-based occlusion queries for fast collision culling between dynamically-generated simplifications of the original models.

We also introduce a new conservative collision error metric. Based on this error metric, we compute the mesh simplification and perform overlap tests between the bounding volumes and the primitives. Our overall algorithm is conservative and never misses any collisions between the original model, though it may return "false positive" collisions within an error bound. Moreover, we only load the cluster hierarchy in the main memory and use out-of-core techniques to fetch the progressive meshes at runtime. Our algorithm has been implemented on a commodity PC with an NVIDIA GeForce FX 5950 Ultra GPU and dual 2.5GHz Pentium IV processors and uses a memory footprint of approximately 250MB. It has been used for real-time dynamic simulation between two complex scanned models consisting of 1.7M and 28M triangles and interactive navigation in a CAD environment composed of more than 12 million triangles. Collision queries using our algorithm take about $15 - 40$ milliseconds to compute all the contact regions on these benchmarks. Some of the key benefits of our approach include:

- **Generality:** Our algorithm makes no assumption with respect to model complexity or topological structures. It can also handle "polygon soup" models.
- **Lower memory overhead:** In practice, the CHPM of a model takes $5 - 8$

times less memory than a BVH. Moreover, our out-of-core algorithm uses a small runtime memory footprint.

- **Fast collision queries:** Our dynamic simplification algorithm bounds the size of the front in each hierarchy and computes all contacts between complex models in tens of milliseconds.

- **Error bounded and conservative:** Our algorithm is conservative in the sense that it detects all contacts. It may report "false positive" collisions within a user-specified error bound.

- **Integrated multiresolution representation:** The dynamic LOD reduces popping in simulation and the CHPM can also be used for interactive display of massive model as described in Chapter 4. Therefore, this new representation can be adopted for interactive display, real-time interaction, and physical simulation of massive models simultaneously.

Organization: The rest of the chapter is organized in the following manner. We present an overview of our approach and the model representation in Section 5.1. Section 5.2 describes the algorithm to compute the CHPM for mainly collision culling and the error metrics used for model simplification. We present our criteria to perform conservative and multiresolution collision queries in Section 5.3 and the overall collision detection algorithm in Section 5.4. We describe its implementation and performance in Section 5.5 and highlight some of the limitations in Section 5.6. Portions of this chapter are described in (Yoon et al., 2004a).

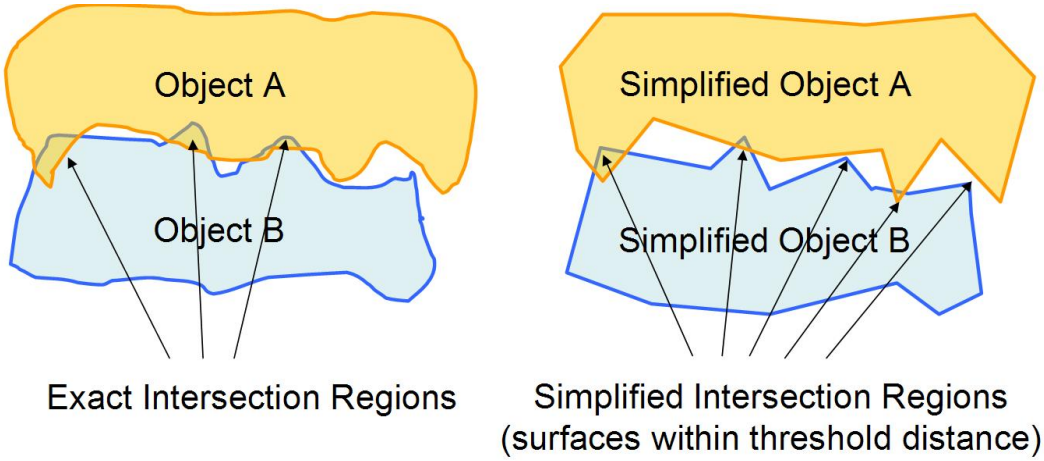


Figure 5.1: **Collision Detection using Dynamic Simplification:** Collision detection between original objects is shown in left and collision between the corresponding simplified objects is shown on the right. All colliding regions between the original objects are detected by our algorithm and we compute a simplified representation of each colliding region. Moreover, "false positive" collisions are also reported within a given error threshold due to the conservativeness of our algorithm.

5.1 Model Representation

In this section we introduce some of the terminology and representations used by our algorithm. We also give a brief overview of our approach for out-of-core hierarchical collision detection.

5.1.1 CHPM Representation

We use a novel representation, a clustered hierarchy of progressive meshes (CHPMs) presented in Section 4.1, for fast collision computation using dynamic simplification of massive datasets. The CHPM representation serves as a dual hierarchy for collision detection: as an LOD hierarchy for error-bounded collision detection and as a bounding volume hierarchy for collision culling. The CHPM consists of two parts: cluster hierarchy and progressive meshes (as shown in Fig. 5.2)

We represent the entire dataset as a hierarchy of clusters, which are spatially lo-

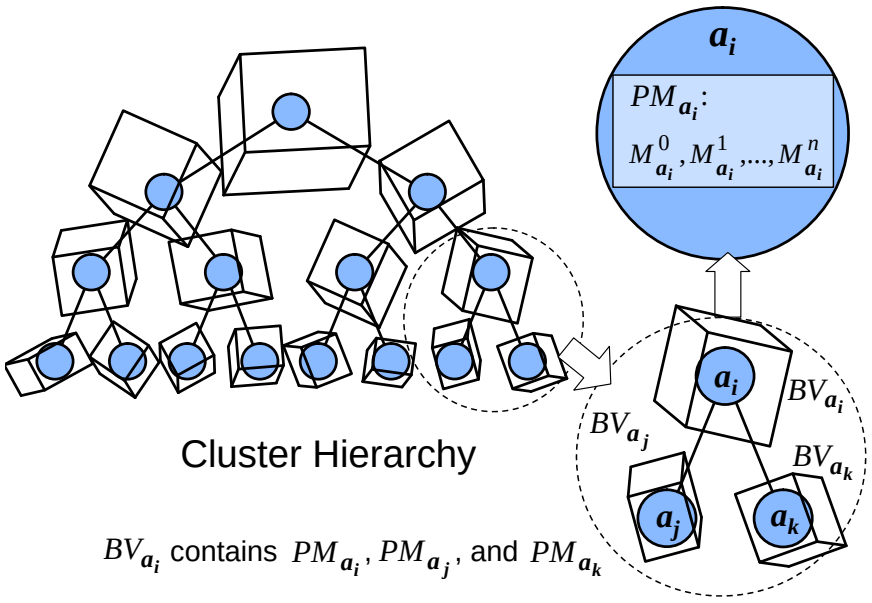


Figure 5.2: **CHPM Hierarchy for Approximate Collision Detection:** We represented the scene as a clustered hierarchy of progressive meshes (CHPM). The CHPM serves as a dual hierarchy: an LOD hierarchy for conservative error-bounded collision and as a bounding volume hierarchy for collision culling. Each cluster contains a progressive mesh and a bounding volume that encloses all geometry in its subtree.

calized mesh regions. As an LOD hierarchy each interior cluster contains a coarser representation of its children's meshes. As a bounding volume hierarchy (BVH) each cluster has an associated bounding volume (BV) which contains all the mesh primitives represented by its subtree. We use the oriented bounding box (OBB) as the BV representation.

Each cluster contains a progressive mesh (PM) as an LOD representation. The PM creates any mesh in the encoded sequences, $M^0, M^1, \ldots, M^n$ by applying vertex splits at runtime. Its detail explanation and notation can be found in Section 4.1.

To detect collisions between a pair of CHPM objects we perform cluster level culling between their cluster hierarchies. Once a set of colliding clusters is computed, PM refinement is performed on and exact collisions between the PM representation are com-

puted. The PMs are used as a continuous LOD representation to alleviate simulation popping artifacts and satisfy the collision error bounds. These two levels of refinement makes CHPMs a middle ground between the flexibility of a vertex hierarchy and the refinement speed of a static LOD (or hierarchical LOD) representation (Erikson et al., 2001).

5.1.2 Dual Hierarchies for Collision Detection

By combining an LOD hierarchy with a traditional BVH we are able to achieve a dramatic acceleration of collision detection between massive models. The CHPM hierarchy allows collisions to be computed using a dynamically generated approximate mesh and thereby reducing the number of overlap tests that need to be performed. Because we use a continuous LOD representation, LOD transitions are smooth and can meet an error bound without being overly conservative.

The collision test between two BVHs can be described by the *bounding volume test tree* (BVTT)(Larsen et al., 2000), a tree structure that holds in each node the result of the query between two BVs. The overall cost of a collision test is proportional to the number of nodes in the front of the BVTT. The basic BVTT algorithm traverses down to the leaves of the BVHs, as long as each query reports a possible collision. However, when traversing the combined cluster hierarchy within the CHPM, the traversal stops when an appropriate LOD is reached. Therefore, the BVTT front size can be dramatically reduced by using LODs and thereby making it possible to perform collision queries between complex models at interactive rates.

Figure 5.3: **Cluster Decomposition of the Lucy Model:** This figures highlights the clusters on the Lucy model (28M triangles). The average cluster size is 1K triangles. Each cluster is represented by a progressive mesh for dynamic simplification and contains a bounding volume for collision culling.

5.2 Simplification and Error Values

An important issue in both mesh simplification for rendering and LOD-based collision detection is the choice of error metrics and their computation. In this section we briefly discuss the CHPM computation algorithm and the error metrics used for conservative error-bounded collision detection.

5.2.1 CHPM Computation for Conservative Collision Culling

Out-of-core clustering and simplification of CHPM computation are well described in Section 4.2. An example of the cluster decomposition is shown in Fig. 5.3.

For conservative collision detection, we use the cluster hierarchy as a BVH and compute an OBB that encloses all the cluster triangles. Moreover, we ensure that the OBB not only encloses the triangles contained in that cluster, but also its descendant

clusters. To guarantee this property each BV is computed as follows: after constructing a PM for the cluster, we use the covariance matrix algorithm (Gottschalk et al., 1996) to compute an OBB that contains all the vertices of the PM. To ensure that all the vertices of the descendant clusters are also contained, each dimension of the OBB is expanded by the maximum surface deviation between the base mesh of the PM and the original mesh.

5.2.2 Conservative Error Metric

Our collision detection algorithm dynamically computes a simplification of each model and checks for collisions between the simplified models. The accuracy of the algorithm is governed by the error function used to compute the simplification. An example of collision detection between simplified objects is shown in Fig. 5.1.

Given two original models, A^0 and B^0, and a minimum separation distance δ, a collision detection algorithm evaluates a function $Collide(A^0, B^0, \delta)$ that computes a set of triangle pairs (t_{A^0}, t_{B^0}) such that $t_{A^0} \in A^0$, $t_{B^0} \in B^0$, and $dist(t_{A^0}, t_{B^0}) < \delta$. For conservative LOD-based collision detection we modify this query. Instead, given the CHPM representations, A and B, we compute:

LodCollide(A, B, δ, ϵ): Determines all pairs (t_A, t_B) such that $t_A \in A$, $t_B \in B$, and $dist(t_A, t_B) < \delta$ with allowed error ϵ, or $dist(t_A, t_B) < (\delta + \epsilon)$. The dynamic simplification used for LOD-based collision detection is determined by the user-specified error ϵ.

Note that this query is defined so that we compute all the triangle pairs within distance $(\delta + \epsilon)$. Thus, our algorithm is a conservative algorithm which will not miss any collisions. We also use another proximity query in our algorithm:

ConservBVTest$(BV_i, BV_j, \delta, \epsilon)$: Given two bounding volumes, BV_i and BV_j, this query conservatively determines whether the subset of the original model contained in

these BVs are colliding (Sec. 5.3).

Many error metrics have been proposed for approximate collision detection, including object size, object velocity, and constant frame-rate for time-critical collision detection (Hubbard, 1993; Otaduy & Lin, 2003; O'Sullivan & Dingliana, 2001). Our simplification algorithm is based on the maximum deviation error or the Hausdorff distance between the original mesh and the simplified mesh, M, denoted $h(M)$. By assuring that the total Hausdorff distance in regions of collision is less than the error threshold, ϵ, we can bound the simulation error. Other collision error metrics based on object size and velocity can be derived from the maximum deviation error (Otaduy & Lin, 2003). In order to perform collision culling between cluster pairs at the cluster level using the CHPM representation, we also store the directed Hausdorff distance between each BV and the original mesh, $\hat{h}(BV)$.

A feature of the Hausdorff metric is that it adapts to the mesh in a contact-dependent manner. The contact forces computed will be more sensitive to simplification in areas with sharp features. However, simplification will be more restricted in such areas because of high deviation in the Hausdorff metric. In relatively flat regions, where the contact forces will be least affected by the simplification, the Hausdorff metric allows greater simplification (Otaduy & Lin, 2003).

5.3 Conservative Collision Formulation

In this section we present our conservative collision scheme which is used to guarantee that a query result using the CHPM representation does not miss any collision as compared to an exact test on the original meshes within the distance error bound, ϵ. In Table 5.1, we highlight the notation used in the rest of the chapter.

In performing LOD-based collision detection we take advantage of the fact that

Notation	Meaning
$\boldsymbol{a}$	A cluster of object A
$PM_{\boldsymbol{a}} = (M^0_{\boldsymbol{a}}, M^1_{\boldsymbol{a}}, ..., M^n_{\boldsymbol{a}})$	The PM of cluster $\boldsymbol{a}$
$h(M^i_{\boldsymbol{a}})$	The directed Hausdorff distance between $M^i_{\boldsymbol{a}}$ and the original mesh
$\hat{h}(BV)$	The directed Hausdorff distance between a bounding volume, BV, and the original mesh
δ	The minimum separation distance for the global collision query. Triangles separated by less than this distance are in collision.
ϵ	The simplification error used for collision detection, specified as a directed Hausdorff distance
$dilate(BV, r)$	An operation that dilates a BV by distance r

Table 5.1: **Notation.** This table highlights the notation used in the rest of the chapter.

CHPM represents a dual hierarchy. $LodCollide()$ can be computed by performing a BVTT traversal between the BVHs of A and B, but a test is needed to check whether the original mesh regions represented by clusters $\boldsymbol{a}$ and $\boldsymbol{b}$ are within distance $\delta + \epsilon$.

The $ConservBVTest()$ query relies on a dilated BV test that is applied to cluster BVs during BVTT traversal and performs overlap tests between the triangles of the PM.

5.3.1 Conservative Collision Metric

We transform the problem of checking whether the original meshes contained inside two BVs are within distance δ into an intersection test between the dilated BVs. Initially, consider the dilated OBB, $dilate(BV, d)$, to be defined as the Minkowski sum of BV with a sphere of radius d and represented as $BV \oplus d$. We use the following lemmas to check whether the original meshes contained inside two bounding volumes, BV_i and BV_j, are within distance $\delta + \epsilon$.

Lemma 1**:** *If the dilated BVs, $dilate(BV_i, \delta/2)$ and $dilate(BV_j, \delta/2)$, do not intersect, the distance between the original meshes contained in the two BVs is greater than δ.*

Proof: Because each BV fully contains a portion of the original mesh, the minimum distance between the two meshes contained in the BVs is at least the sum of dilation amounts, δ.

Lemma 2**:** *If there is an intersection between dilated BVs $dilate(BV_i, \delta/2)$ and $dilate(BV_j, \delta/2)$ the distance between the original meshes contained in the BVs has an upper bound of $\delta + \hat{h}(BV_i) + \hat{h}(BV_j)$.*

Proof: Due to the conservativeness of the BVs, the BVs may intersect even though the meshes may not be colliding. By definition of directed Hausdorff distance, every point of each original BV is within distance $\hat{h}(BV)$ of the original mesh. Furthermore, the dilated

BVs are within distance $\delta/2$ of the original BV. Therefore, the maximum total distance between the original meshes is $\delta/2 + \hat{h}(BV_i) + \delta/2 + \hat{h}(BV_j) = \delta + \hat{h}(BV_i) + \hat{h}(BV_j)$.

These Lemmas lead directly to the definition of $ConservBVTest()$:

$$ConservBVTest(BV_i, BV_j, \delta, \epsilon) = \begin{cases} NoCollision, & \neg isect(dilate(BV_i), dilate(BV_j)) \\ Collision, & isect(dilate(BV_i), dilate(BV_j)) \\ & \text{and } \hat{h}(BV_i) + \hat{h}(BV_j) \leq \epsilon \\ PotentialCollision, & isect(dilate(BV_i), dilate(BV_j)) \\ & \text{and } \hat{h}(BV_i) + \hat{h}(BV_j) > \epsilon \end{cases}$$

where *isect* is a bounding volume intersection test and the shorthand $dilate(BV)$ simply indicates $dilate(BV, \delta/2)$.

If the dilated boxes do not intersect then we know that the original meshes are not colliding by Lemma 1. However, if these boxes overlap we use the Hausdorff distances $\hat{h}(BV_i)$ and $\hat{h}(BV_j)$ to determine whether we can conclude that the original models are colliding. When $\hat{h}(BV_i) + \hat{h}(BV_j) \leq \epsilon$ then by Lemma 2 we can conclude that the distance between the original meshes must be within $\delta + \epsilon$.

Rather than computing the exact Minkowski sum, we instead compute $dilate(BV, d)$ as an approximation of $BV \oplus d$ by extending each dimension of the OBB by $d/2$ from the center of the OBB. To satisfy Lemma 2, the $\hat{h}$ value associated with BV is extended by the maximum deviation between $dilate(BV, d)$ and $BV \oplus d$.

5.3.2 Cull and Refine Operations

To compute $LodCollide()$ we first refine the mesh for each object such that the sum of mesh deviations of each model is less than ϵ in regions of collision. Next, we check whether the selected LOD representations are within distance δ. Both parts of this computation use the $ConservBVTest()$ query through two operations:

- **Culling operation:** BV pairs whose distance is greater than δ are culled. To conservatively perform this culling step, we apply the $ConservBVTest()$ test by dilating the BVs of the two approximate mesh portions and checking for intersection between the dilated BVs. BVs for which $ConservBVTest()$ finds no collisions cannot be intersecting and are culled away.

- **Refining operation:** $ConservBVTest()$ can determine when the LOD resolution must be increased. The BV pairs, for which the $ConservBVTest()$ query reports a collision, has total simplification error less than ϵ and the triangles within the BVs are in collision. On the other hand, when $ConservBVTest()$ reports a potential collision the total Hausdorff distance is too high and further refinement needs to be performed on one of the BVs. We guarantee that refinement always decreases the $\hat{h}$ values [1]. Once the total Hausdorff distance is below ϵ, $ConservBVTest()$ becomes an exact collision test.

By recursively performing these two operations, we can compute the triangle pairs from dynamic LODs whose distance is less than δ. More importantly, their counterparts in the original meshes are separated by less than $\delta + \epsilon$.

5.4 Fast Collision Detection

In this section, we present a hierarchical collision detection algorithm based on the CHPM. We also present several culling techniques to improve its performance.

---

[1] If the $\hat{h}$ of a BV is smaller than the maximum $\hat{h}$ of its child's BVs during the construction of our representation, we set $\hat{h}$ of the BV as the maximum of the $\hat{h}$.

5.4.1 Overall Algorithm

The overall algorithm for collision detection between two CHPM objects is shown in Alg. 1. We compute the colliding front of the bounding volume test tree (BVTT) using the culling and refining operations presented in Sec. 5.3.2. The colliding front contains pairs of clusters from the two objects that are in collision. For each of these cluster pairs, we perform an exact collision test after refining their PMs. This provides the fine-grained control of the simulation error. The cluster collision test uses a further collision culling algorithm based on 2.5D overlap tests that relies on GPU visibility queries. Exact collision tests are performed after this additional culling step.

Algorithm 1 Compute collisions between two objects ($LodCollide()$)
Input: A, B: Objects' δ: min. separation distance; ϵ: LOD error bound
Output: triangles of A and B in collision

```
LodCollide(A,B,δ,ϵ)
  tris ← ∅
  Front ←ComputeBVTTFront(A,B,δ,ϵ)
  for all (a, b) ∈ Front do
    tris ← tris∪ClusterCollide(a,b,δ,ϵ)
  end for
  return tris
```

5.4.2 Bounding Volume Test Tree (BVTT)

We use the concept of the bounding volume test tree (BVTT)(Larsen et al., 2000) to accelerate the computation of $LodCollide()$. In the CHPM representation, the cluster hierarchy is also a BVH. We traverse the BVHs of both the objects and compute the BVTT.

A node $(\boldsymbol{a}, \boldsymbol{b})$ in the BVTT represents a test between clusters $\boldsymbol{a}$ and $\boldsymbol{b}$ from objects A and B, respectively. If the test determines that the objects are non-colliding then the node is a leaf of the BVTT and no further tests are needed between the subtrees of A and B rooted at $\boldsymbol{a}$ and $\boldsymbol{b}$. Otherwise, there is a potential collision between $\boldsymbol{a}$ and $\boldsymbol{b}$.

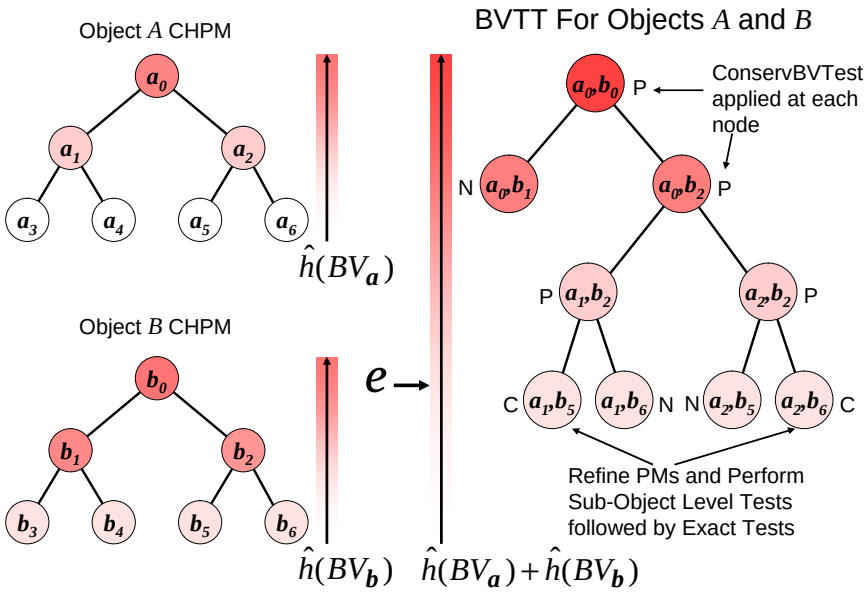


Figure 5.4: **BVTT.** Each node of the bounding volume test tree (BVTT) represents a test between a cluster from each of two colliding objects. The test, $ConservBVTest()$, uses the clusters' bounding volumes to determine whether the cluster pair is not colliding (N), colliding (C), potentially colliding (P). The distinction between colliding and potentially colliding depends upon the sum of the clusters' associated errors (indicated by error bars) being below the error-bound, ϵ.

If the total Hausdorff error of $\boldsymbol{a}$ and $\boldsymbol{b}$, $(\hat{h}(BV_{\boldsymbol{a}}) + \hat{h}(BV_{\boldsymbol{b}}))$, is less than ϵ, an exact test is performed to determine the triangles in collision; otherwise the cluster with greater error is refined (see Fig. 5.4). As shown in Alg. 2, we use the $ConservBVTest()$ query to traverse the hierarchies of A and B, which implicitly computes the BVTT. The BVTT traversal effectively performs coarse-grained LOD refinement by selecting the clusters from objects A and B used for exact collision detection.

CHPM Front Computation

The BVTT front computed in the algorithm described above may contain multiple clusters representing the same portion of either A or B. This situation occurs when the traversal reaches BVTT nodes such as $(\boldsymbol{a_1}, \boldsymbol{b_1})$ and $(\boldsymbol{a_1}, \boldsymbol{b_2})$. It may be the case

Algorithm 2 Perform BVTT traversal and compute the colliding BVTT front
Input: A, B: Objects, δ: min. separation distance, ϵ: LOD error bound
Output: triangles of A and B in collision

```
ComputeBVTTFront(A,B,δ,ϵ)
  return BVTest(Root(A),Root(B),δ, ϵ)

BVTest(a,b,δ,ϵ)
  t ← ConservBVTest(BV_a, BV_b, δ, ϵ)
  if t = NoCollision then
    {Culling: contained original meshes are further than δ}
    return ∅
  else if t = Collision then
    {Bounding boxes in collision, total error is less than ϵ}
    {These nodes are part of the colliding front}
    return (a,b)
  else {t = PotentialCollision}
    {Refining: total error is greater than ϵ}
    if ĥ(BV_a) > ĥ(BV_b) then
      return BVTest(LeftChild(a), b, δ, ϵ)
      ∪ BVTest(RightChild(a)), b, δ, ϵ)
    else
      return BVTest(a, LeftChild(b), δ, ϵ)
      ∪ BVTest(a, RightChild(b), δ, ϵ)
    end if
  end if
```

that $\hat{h}(BV_{\boldsymbol{a_1}}) + \hat{h}(BV_{\boldsymbol{b_1}}) > \epsilon$ but $\hat{h}(BV_{\boldsymbol{a_1}}) + \hat{h}(BV_{\boldsymbol{b_2}}) \leq \epsilon$. The traversal will split $\boldsymbol{a_1}$ into $\boldsymbol{a_2}$ and $\boldsymbol{a_3}$ in one branch of the BVTT but $\boldsymbol{a_1}$ will fall on the BVTT front in the other branch. We would like to have a single unique front across each CHPM. In order to maintain this property the BVTT node $(\boldsymbol{a_1}, \boldsymbol{b_2})$ is forced to split into nodes $(\boldsymbol{a_2}, \boldsymbol{b_2})$ and $(\boldsymbol{a_3}, \boldsymbol{b_2})$.

Coherence-Based BVTT Front Computation

A further modification of the algorithm described above is made to take advantage of temporal coherence. Rather than recursively computing the BVTT front from the root for each timestep, we traverse the front from the previous timestep and make incremental updates. By collapsing the BVTT nodes into their parent node the level of refinement is reduced, and by splitting a BVTT node the level of refinement is increased. This approach leads to up to two times reduction of the time spent on BVTT computation.

Algorithm 3 Compute collision between two clusters
Input: $\boldsymbol{a}$, $\boldsymbol{b}$: clusters, δ: min. separation distance, ϵ: LOD error bound
Output: triangles of A and B in collision

```
ClusterCollide(a,b,δ,ε)
  RefinePMs(PM_a,PM_b,ε)
  T ← SubObjectCull(a,b,δ) {T is a set of triangle pairs}
  return ExactTest(T,δ)
```

5.4.3 Computing Dynamic LODs

We process each pair of clusters, $(\boldsymbol{a}, \boldsymbol{b})$, on the colliding front of the BVTT for exact collision detection. As shown in Alg. 3, the first step is to refine the PMs of the clusters. Each cluster pair must have a total deviation from the original meshes of not more than ϵ. In order to take advantage of temporal coherence, we refine the PMs based on their current state. If the sum of the errors is greater than ϵ, we apply vertex-splits

to the PM with greater error until the error falls below ϵ. If the sum of errors is less than ϵ, we apply edge-collapses to the PM with lower error until applying one more edge-collapse would cause the total error to exceed ϵ. Once the PMs are refined, the total simplification error at each point of contact between the clusters will be less than ϵ. Since a single cluster may be in multiple cluster pairs of the BVTT front we ensure that the PMs are refined to meet the error bound in each BVTT front node.

5.4.4 GPU-based Culling

Performing all $O(n^2)$ pairwise tests between triangles of two clusters can be an expensive operation as the clusters may contain around 1K triangles. To further reduce the potentially colliding set of triangles, we employ GPU-based culling similar to (Govindaraju et al., 2003b; Govindaraju et al., 2004). Triangles in the mesh selected from each cluster's PM are randomly partitioned into "sub-objects" of size k triangles. For each triangle of a sub-object we construct a BV dilated by $\delta/2$. Since these BVs must be constructed quickly at runtime, we use axis aligned bounding boxes.

We use GPU-based occlusion queries to cull the sub-objects between the two clusters. After rendering some geometric primitives, an occlusion query returns the number of pixels that pass the depth buffer test. We use these queries to perform a 2.5D overlap test between bounding volumes along the three orthogonal axes. First, the BVs for all the triangles of the first cluster are rendered under an orthographic projection. Then, the BVs for sub-objects from the second cluster are rendered with the depth test set to GL_GEQUAL. Sub-objects of the second cluster that have no pixels pass this reversed depth test are classified as non-intersecting with the BVs of all objects of the first cluster. These sub-objects may be culled from the set of possible collisions. The test is performed for projections along the x, y, and z axes. The same test is performed with the order of the clusters switched to cull sub-objects of the first cluster.

In order to ensure that errors are not introduced due to sampling in the frame buffer, we use a conservative algorithm to perform GPU-based culling (Govindaraju et al., 2004). The BVs are expanded by taking their Minkowski sum with a sphere to ensure that they are rasterized into every pixel which they may partially cover.

5.4.5 Triangle Collision Test

We perform exact collision detection for triangles pairs that pass sub-object culling. Each triangle in the LOD representation of an object represents a set of triangles of the original model. In order to conservatively meet the error bound, an OBB is constructed for each triangle that contains the triangle plus the original mesh triangles that were simplified into it. To enclose the original geometric primitives, the OBB is initially a flat box aligned with the plane of the triangle containing its vertices. It is then dilated by the $\hat{h}$ value of its cluster. The OBBs are then further dilated by $\delta/2$ before being tested for intersection. Triangles whose enclosing OBBs are overlapping are reported as colliding.

5.4.6 Out-of-Core Computation

Our goal is to perform collision detection between models that cannot be stored in main memory. The CHPM representation also serves as a mechanism for out-of-core management. At runtime we keep the CHPM hierarchy for each object in the main memory, while the PMs for each cluster reside on the disk. A working set of PMs is kept in memory for collision detection. For each pair of colliding objects, we keep PMs for nodes on the BVTT front in main memory as well as their parents and children to handle LOD switches.

5.4.7 Unified Multiresolution Representation

One advantage of our approach is that the dynamic LOD representation used for collision detection can also be used for interactive rendering, which was described in Chapter 4. This can be especially important for handling massive models. The memory requirements of storing separate representations for collision detection and rendering may be prohibitively high. LOD selection for collision detection and graphical rendering can be unified by appropriate error metrics. When computing the BVTT we stop the traversal only when metrics for both collision tests and visual rendering have been satisfied. Similarly, the PMs are refined so that the LOD error is less than the error bounds for both collision detection and visual rendering.

5.5 Implementation and Performance

In this section we describe our implementation and highlight its performance on complex models.

5.5.1 Implementation

We have implemented our out-of-core simplification and runtime system on a dual 2.4GHz Pentium-IV PC, with 1GB of RAM and a GeForce FX 5950 Ultra GPU with 128MB of video memory. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files for out-of-core access to the data.

We achieve high throughput for rendering and sub-objects culling from graphics cards by storing the mesh data on the GPU, thereby reducing the data transferred to the GPU each frame. We use the GL_ARB_vertex_buffer_object OpenGL extension that performs GPU memory management for both the vertex and the face arrays.

Model	Lucy	PP	Turbine	Dragon
Triangles (M)	28	12.8	1.7	0.8
Num Clusters (K)	14	6.4	3.4	1.7
Size of CHPM (MB)	1341	849	88	48

Table 5.2: **Benchmark Models:** Model complexity and number of cluster are shown.

Each timestep we only to need to update the BVs and mesh data of clusters whose PMs have changed refinement level since the previous timestep. Furthermore, we use GL_NV_occlusion_query extension to perform collision culling.

5.5.2 Benchmark Models

Our algorithm has been applied to two different applications with massive models. They are :

- **Dynamic simulation:** A Lucy model falling onto the CAD model of a turbine blade.
- **Navigation:** A user navigating in a coal-fired power plant model with a flying dragon model.

The Lucy model composed of more than 28 million polygons (Fig. 1.5), the power plant consisting of more than 12 million polygons and 1200 objects (Fig. 5.5), the CAD turbine model consisting of a single 1.7-million polygon object (Fig. 1.5), and the dragon model consisting of 800 thousand polygons (Fig. 5.5). The details of these models are shown in Table 5.2.

5.5.3 Performance

Dynamic simulation: We have implemented an impulse based rigid body simulation(Mirtich & Canny, 1995). We are able to perform collision detection between the

Lucy and blade model at an interactive rate (12-30 frames per second). An image sequence from this simulation is show in Fig. 1.5. The average collision query time was 18ms. Moreover, we are able to simultaneously perform interactive rendering and collision detection by using a 250MB memory footprint. Most of the query time is spent on the sub-object culling (55%) and very little is spent on PM and cluster refinement (1%).

Navigation: For our navigation benchmark we moved a 0.8M triangle dragon model along a path in the 12M triangle power plant model and detected collisions with the objects in the power plant model. Fig. 5.5 shows a screenshot from the path. The average query time is 55ms and the memory footprint is 200MB.

5.5.4 Memory requirement

Our CHPM as a dual hierarchy requires 122MB per million vertices on average. Quantization for geometry and compression on PMs can further improve the memory requirement This is low compared to around 560MB per million vertices needed to represent an OBBtree (Gottschalk et al., 1996). Furthermore, our out-of-core representation keeps only the cluster hierarchy and the PMs of a subset of the clusters in the main memory.

5.6 Analysis and Limitation

In this section, we briefly discuss factors that affect the performance of our algorithm and its limitations.

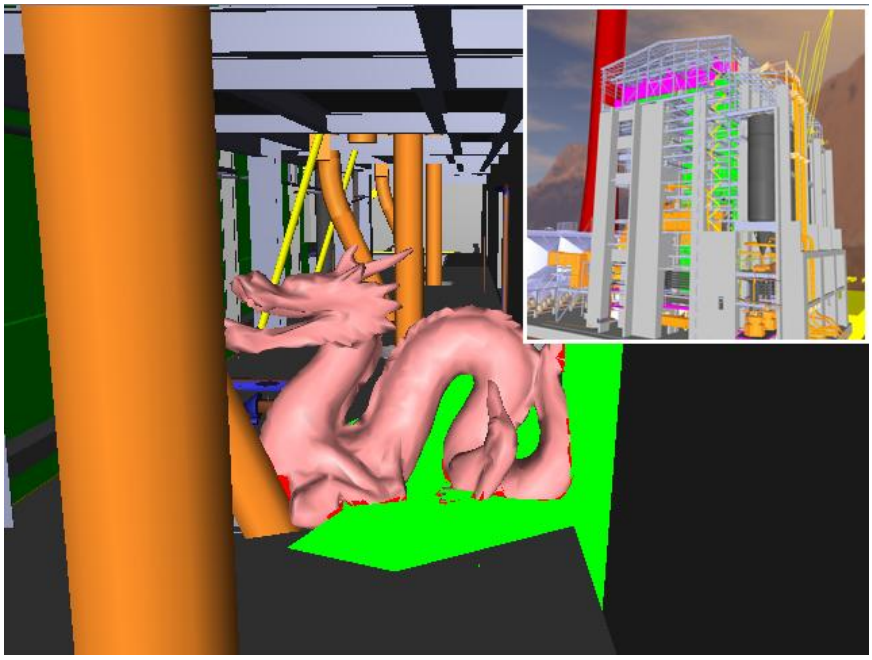


Figure 5.5: **Collision Example.** We tested our conservative collision detection algorithm on a path for the 0.8M triangle dragon model along a path through the 12M-triangle power-plant model. The average collision query time is 55ms and the total memory footprint is 200MB. The error bound is set to 0.04% of the width of the dragon model. In this path the models have deep penetration and this screenshot colliding triangles are show in red and green. In the upper right the entire power plant model is shown to illustrate its complexity.

5.6.1 Performance Analysis

The performance of our algorithm depends on many factors including motion coherence, relative contact configuration, model tessellation, and the error bound, ϵ. In general, our algorithm achieves the highest speed-up in regions of contact between highly-tessellated and almost flat surfaces. In such regions, the algorithm computes a drastic simplification with a low Hausdorff deviation. Furthermore, the OBBs fit flat mesh regions more tightly than those regions with high curvature.

Our algorithm also exploits temporal and spatial coherence between successive frames. The coarse-grained cluster level refinement performs incremental computations to refine the front. The out-of-core management relies on coherence between timesteps to fetch and prefetch PMs from the disk. We also exploit coherence to reuse bounding box data loaded into the GPU memory, which is needed to obtain high throughput from the GPUs for occlusion queries.

5.6.2 Comparison with CLODs

CLODs proposed by Otaduy and Lin [OL03] are precomputed dual hierarchies of static LODs used for multiresolution collision detection. The precomputed LODs and their bounding volume hierarchies are used to accelerate collision computations at runtime. As a result, the runtime overhead of CLODs is relatively small as compared to our approach. However, switching LODs between static LODs in the CLOD-algorithm can result in a large discontinuity in the simulation. On the other hand, our algorithm provides smooth fine-grained local control of simplification error within each cluster. This operation is very efficient and reduces the "popping" or discontinuities between successive collision queries. The underlying formulation of CLODs assumes that the input model is a closed, manifold solid and is not applicable to polygon soups. On the

other hand, our algorithm is applicable to all models, including polygon soups [2].

5.6.3 Limitations

Our algorithms works well for our current set of applications. However, it has some limitations. It relies on temporal coherence for out-of-core management, front computation, and GPU memory management. In situations where many objects come into close proximity within a single timestep, memory stalls may occur as PMs are fetched from the disk. Also, if there is little motion coherence between successive instances then fetching for out-of-core may not keep up with the simulation. Moreover, our algorithm can be very conservative in some cases. Our surface deviation error bounds may not be very tight for certain meshes. Moreover, our algorithm can be overly conservative and may return too many "false positives." An example is two objects (e.g. two concentric spheres) in parallel close proximity with a separation distance, d, where $\delta < d < \delta + \epsilon$.

[2]For our CHPM representations of polygon soups, we can use vertex clustering operations as simplification operations.

Chapter 6

Cache-Coherent Layouts

A major computing trend over the last few decades has been the widening gap between processor speed and main memory speed. For example, CPU performance has increased 60% per year for nearly two decades. On the other hand, the main memory and disk access time only decreased by 7–10% per year during the same period (Ruemmler & Wilkes, 1994; Patterson et al., 1997). A relative performance gap between CPU performance and access time to DRAM is shown in Fig. 6.1. As a result, system architectures increasingly use caches and memory hierarchies to avoid memory latency. The access times of different levels in a memory hierarchy typically vary by orders of magnitude. In some cases, the running time of a program is as much a function of its cache access pattern and efficiency as it is of operation count (Frigo et al., 1999; Sen et al., 2002).

Our goal is to design cache efficient algorithms to process large meshes for a wide variety of applications including dynamic simplification for interactive view-dependent rendering and collision detection. The two standard techniques used to reduce cache misses are:

1. **Computation Reordering**: Reorder the computation to improve program locality. This is performed using compiler optimizations or application specific hand-tuning.

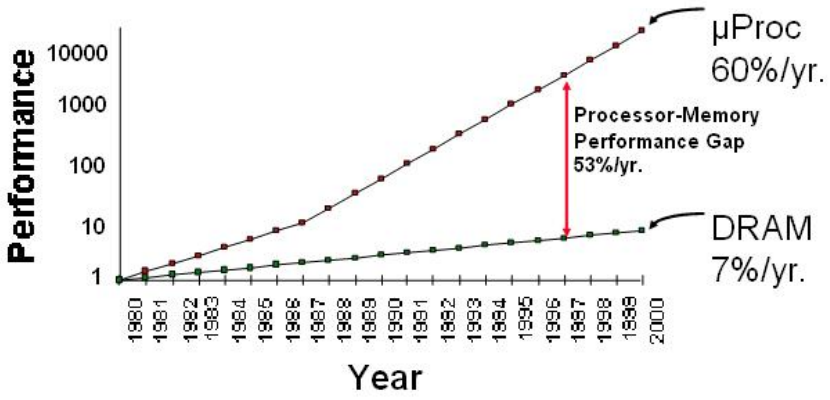


Figure 6.1: **Relative Performance Gap between CPU Processing Power and Access Time to Main Memory:** The CPU performance has increased 60% per year for almost two decades. On the other hand, the access time for main memory consisting of DRAM only decreased by 7-10% per year during the same period. The graph shown is excerpted from a talk slide of Trishul Chilimbi.

2. **Data Layout Optimization:** Compute a cache-coherent layout of the data in memory according to the access pattern.

In this chapter, we focus on data layout optimization of large meshes to improve cache coherence. A triangle mesh is represented by linear sequences of vertices and triangles. Therefore, the problem becomes one of computing a cache efficient layout of the vertices and triangles.

Many layout algorithms and representations have been proposed for optimizing the cache access patterns for specific applications. The representations include *rendering sequences* (e.g. triangle strips) that are used to improve the rendering performance of large meshes on GPUs. Recent extensions include *processing sequences* (e.g. streaming meshes), which work well for applications that can access the data in a fixed order. However, many applications do not have a fixed processing order; the application of the processing sequences is limited. Some algorithms for image processing and visualization of large datasets use space filling curves as a heuristic to improve cache coherence of a layout. These algorithms work well on models with a regular structure. However,

they do not take into account the topological structure of a mesh and are not general enough to handle unstructured datasets.

Main Results: We present a novel method to compute cache-oblivious layouts of large triangle meshes. Our approach is general in terms of handling all kinds of polygonal models. Also, it is cache-oblivious, as it does not require any knowledge of the cache parameters or block sizes of the memory hierarchy involved in the computation. We only assume that runtime applications access the large triangle meshes in a cache-coherent manner, which is the case in many geometric processing applications including visualization, collision detection, and isosurface extraction.

We represent the mesh as an undirected graph $G = (V, E)$, where $|V| = n$ is the number of vertices. The mesh layout problem is reduced to computing an optimal one-to-one mapping of vertices to positions in the layout, $\varphi : V \rightarrow \{1, ..., n\}$, that reduces the number of cache misses. Our specific contributions include:

1. Deriving a practical cache-oblivious metric that estimates the number of cache misses.

2. Transforming the layout computation to an optimization problem based on our metric.

3. Solving the combinatorial optimization problem using a multilevel minimization algorithm.

We also extend our graph-based formulation to compute cache-oblivious layouts of bounding volume and multiresolution hierarchies of large meshes.

We use cache-oblivious layouts for three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. In order to show the generality of our approach, we compute layouts of several kinds

of geometric models. These include CAD environments, scanned models, isosurfaces, and terrains. We directly use these layouts without any modification to the runtime application. Our layouts significantly reduce the number of cache misses and improve the overall performance. Compared to a variety of popular mesh layouts, we achieve on average:

1. Over an order of magnitude improvement in performance for isocontour extraction.
2. A five time improvement in rendering throughput for view-dependent rendering of multi-resolution meshes.
3. A two time speedup in collision detection queries based on bounding volume hierarchies.

Organization: The rest of the chapter is organized as follows. Section 6.1 gives an overview of our approach and presents techniques for computing the graph layout of hierarchical representations. We present our cache-oblivious metric in Section 6.2 and then describe the multilevel optimization algorithm for computing the layouts in Section 6.3. Next, Section 6.4 highlights the use of our layouts in three different applications. Finally, we analyze our algorithms and discuss some of their limitations in Section 6.5. Portions of this chapter are described in (Yoon et al., 2005a).

6.1 Mesh Layout and Cache Misses

In this section, we introduce some of the terminology used in the rest of the chapter and give an overview of memory hierarchies. We represent a mesh as a graph and extend our approach to layouts of multi-resolution and bounding volume hierarchies of a mesh.

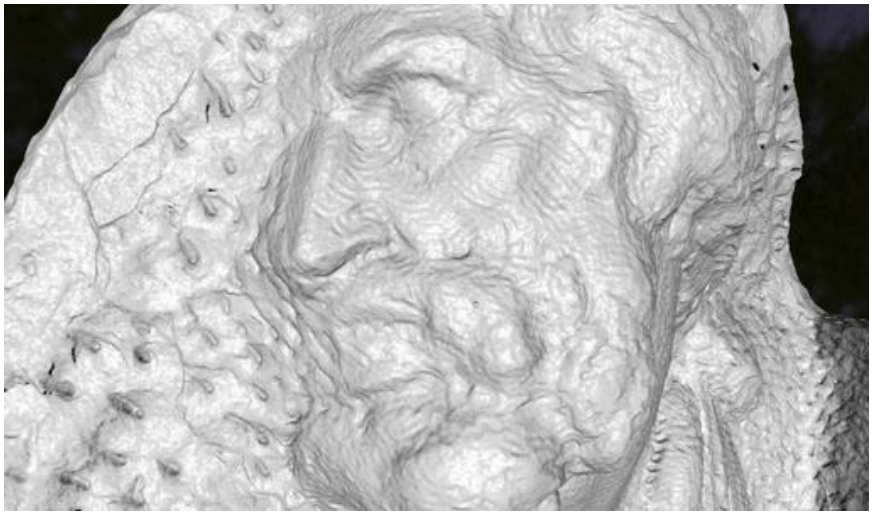


Figure 6.2: **Scan of Michelangelo's St. Matthew:** We precompute a cache-oblivious layout of this 9.6GB scanned model with 372M triangles. Our novel metric results in a cache-oblivious layout, which at runtime reduces the vertex cache misses by more than a factor of four for interactive view-dependent rendering. As a result, we improve the frame rate by almost five times. We achieve a throughput of 106M tri/sec (at 82 fps) on an NVIDIA GeForce 6800 GPU.

6.1.1 Memory Hierarchy and Caches

Most modern computers use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. Memory hierarchies have two main characteristics. First, higher levels are larger in size and farther from the processor, and they have slower access times. Second, data is moved in large blocks between different memory levels. The mesh layout is initially stored in the highest memory level, typically the disk. The portion of the layout accessed by the application is transferred in large blocks into the next lower level, such as main memory. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. The number of cache misses is dependent on the layout of the original mesh in memory and the access pattern of the application.

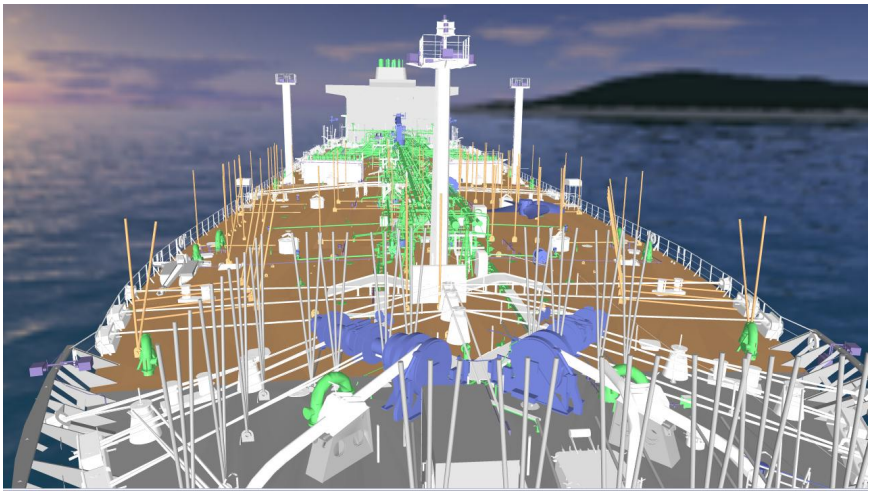


Figure 6.3: **Double Eagle Tanker:** We compute a cache-oblivious layout of the tanker with 82M triangles and more than 127K different objects. This model has an irregular distribution of primitives. We use our layout to reduce vertex cache misses and to improve the frame rate for interactive view-dependent rendering by a factor of two; we achieve a throughput of 47M tri/sec (at 35 fps) on an NVIDIA GeForce 6800 GPU.

6.1.2 Mesh Layout

A mesh layout is a linear sequence of vertices and triangles of the mesh. We construct a graph in which each vertex represents a data element of the mesh. An edge exists between two vertices of the graph if their representative data elements are likely to be accessed in succession by an application at runtime.

For a single-resolution mesh layout, we map mesh vertices and edges to graph vertices and edges. A vertex layout of an undirected graph $G = (V, E)$ is a one-to-one mapping of vertices to positions, $\varphi : V \rightarrow \{1, \ldots, n\}$, where $|V| = n$. Our goal is to find a mapping, φ, that minimizes the number of cache misses during accesses to the mesh.

A mesh layout is composed of two layouts: a vertex layout and a triangle layout. We can process the triangle layout in the same manner which we defined and constructed the vertex layout. For the sake of clarity, throughout the remainder of the chapter, we use the term *layout* to refer to a vertex layout. In Section 6.4.1 we discuss efficient

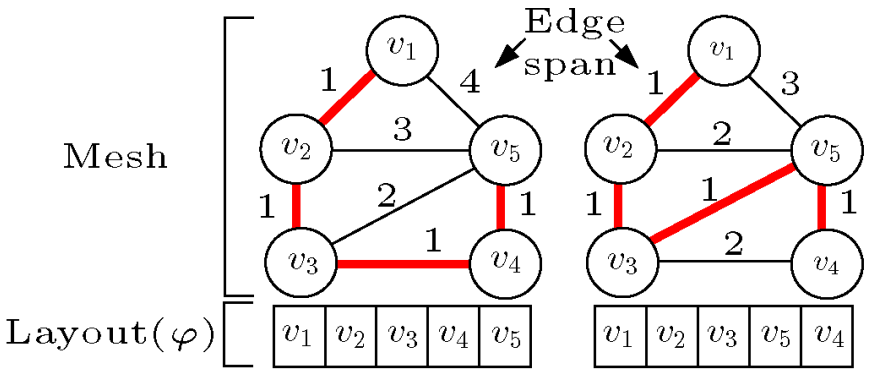


Figure 6.4: **Vertex layout for a mesh:** A mesh consisting of 5 vertices is shown with two different orderings obtained using a local permutation of v_4 and v_5. We highlight the span of each edge based on the layout. The ordering shown on the right minimizes cache misses according to our cache-oblivious metric.

computations of two layouts in more detail.

6.1.3 Layouts of Multiresolution Meshes and Hierarchies

In this section, we show that our graph-based formulation can be used to compute cache-coherent layouts of hierarchical representations. Hierarchical data structures are widely used to speed up computations on large meshes. Two types of hierarchies are used for geometric processing and interactive visualization: bounding volume hierarchies (BVHs) and multi-resolution hierarchies (MRHs). The BVHs use simple bounding shapes (e.g. spheres, AABBs, OBBs) to enclose a group of triangles in a hierarchical manner. MRHs are used to generate a simplification or approximation of the original model based on an error metric; these include vertex hierarchies (VHs) used for view-dependent rendering, and hierarchies that are defined using subdivision rules.

Terminology: We define $v_i = v_i^0$ as the ith vertex at the leaf level of the hierarchy, and v_i^k as a vertex at the kth level. v_i^k is a parent of v_i^{k-1} and v_{i+1}^{k-1}. In the case of a BVH, v_i^k denotes a bounding volume. In the case of a vertex hierarchy, v_i^k denotes a vertex generated by decimation operations. An example of a vertex hierarchy is shown

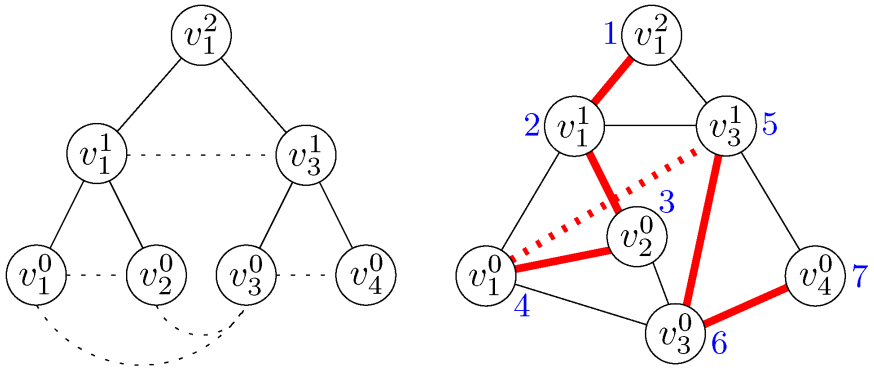


Figure 6.5: **A layout of a vertex hierarchy:** A vertex hierarchy is shown on the left. Each node of the vertex hierarchy represents a leaf or intermediate level vertex. A parent node, v_1^1, is constructed by merging two child nodes, v_1^0 and v_2^0. Solid lines between the nodes represent connectivity access and dotted lines represent the spatial locality between the nodes at the same level. The corresponding graph and a layout of the vertices (with a position in the layout illustrated in blue) are shown on the right.

in Fig. 6.5.

In order to compute a layout of a hierarchy, we construct a graph that captures cache-coherent access patterns to the hierarchy. We add extra edges to our graph in order to capture the spatial locality and parent-child relationships within the hierarchy.

1. **Connectivity between parent-children nodes:** Once a node of a hierarchy is accessed, it is highly likely that its parent or child nodes would also be accessed soon. For example, a vertex-split of a node in the VH activates its child nodes and an edge-collapse of two sibling nodes activates their parent node.

2. **Spatial locality between vertices at the same level:** Whenever a node is accessed, other nodes in close proximity are also highly likely to be accessed thereafter. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is activated, other nearby nodes are either colliding or are in close proximity and may be accessed soon.

Graph Representation: We take these localities into account and compute an undirected graph for MRHs and BVHs. For a BVH, we represent each BV with a separate vertex in the graph. The edges in our graph include edges between parent vertices and their children, as well as edges between nearby vertices at each level of the BVH. Edges are created between nearby vertices when their Euclidean distance falls below a given threshold. Fig. 6.5 shows the graph as well as its layout for the given vertex hierarchy. More details on connectivity and spatial localities of BVHs are also available in the next chapter 7.

6.2 Cache-Oblivious Layouts

In this section we present a novel algorithm for computing a cache-coherent layout of a mesh. We only assume that runtime applications access their layouts in a cache coherent manner. However, we make no assumptions about cache parameters and therefore compute the layout in a cache-oblivious manner.

6.2.1 Terminology

We use the following terminology in the rest of the paper. The *edge span* of the edge between v_i and v_j in a layout is the absolute difference of the vertex indices, $|i - j|$ (see Fig. 6.4). We use E_l to denote the set that consists of all the edges of edge span l, where $l \in [1, n-1]$. The *edge span distribution* of a layout is the histogram of spans of all the edges in the layout. The *cache miss ratio* is the ratio of the number of cache misses to the number of accesses. The *cache miss ratio function (CMRF)*, p_l, is a function that relates the cache miss ratio to an edge span, l. The CMRF always lies within the interval $[0, 1]$; it is exactly 0 when there are no cache misses, and equals 1 when every access results in a cache miss. We alter the layouts using a *local permutation* that

reorders a small subset of the vertices. The local permutation changes the edge span of edges that are incident to the affected vertices (see Fig. 6.4).

6.2.2 Metrics for Cache Misses

We first define a metric for estimating the cache misses for a given layout. One well known metric for the graph layout problem is the minimum linear arrangement (MLA), which minimizes the sum of edge spans (Diaz et al., 2002). Heuristics for the NP-hard MLA problem, such as spectral sequencing, have been used to compute mesh layouts for rendering and processing sequences (Bogomjakov & Gotsman, 2002; Isenburg & Lindstrom, 2005). We have empirically observed that metrics used to estimate MLA may not minimize cache misses for general applications (See Fig. 6.7). This is mostly because MLA results in a front-advancing sweep over the mesh along a dominant direction, which tends to minimize the length of the front. On a rectilinear grid, for example, the optimal MLA layout has two distinct portions: one of them corresponds to a row-by-row layout and the other portion corresponds to a column-by-column layout (Fishburn et al., 2000), which exhibits poor performance when accessing the grid diagonally. We present an alternative metric based on the edge span distribution and the CMRF that captures the locality for various access patterns and results in layouts with an improved "space filling" quality. Contrary to MLA, our layouts are not biased towards a particular traversal direction.

Cache-coherent Access Pattern: If we know both the runtime access pattern of a given application a priori and the CMRFs, we can compute the exact number of cache misses. However, we make no assumptions about the application and instead use a probabilistic model to estimate the number of cache misses. Our model approximates the edge span distribution of the runtime access pattern of the vertices with the edge

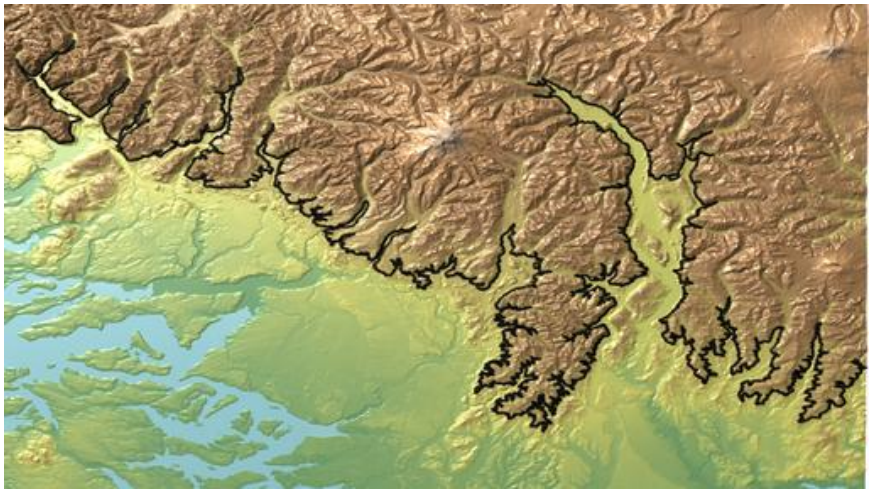


Figure 6.6: **Puget Sound contour line:** This image shows a contour line (in black) extracted from an unstructured terrain model of the Puget Sound. The terrain is simplified down to 143M triangles. We extracted the largest component (223K edges) of the level set at 500 meters of elevation. Our cache-oblivious layouts improve the performance of the isocontour extraction algorithm by more than an order of magnitude.

span distribution of the layout. Based on this model, we define the expected number of cache misses of the layout as:

$$ECM = \sum_{i=1}^{n-1} |E_i| p_i \tag{6.1}$$

where $|E_i|$ is the cardinality of E_i and is a function of the layout, φ.

6.2.3 Assumptions

Our goal is to compute a layout, φ, that minimizes the expected number of cache misses for all possible cache parameters. We present a metric that is used to check whether a local permutation would reduce cache misses. We make two assumptions with respect to CMRFs: invariance and monotonicity.

Invariance: We assume that the CMRF of a layout is invariant before and after a local permutation. Since a local permutation affects only a small region of a mesh, the

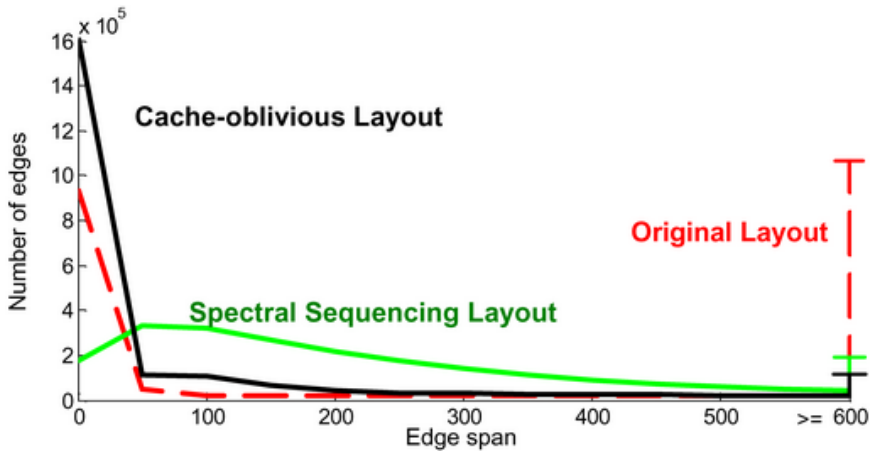


Figure 6.7: **Edge span distributions:** The edge span histogram of the dragon model with 871K triangles and 437K vertices. We show the histogram of the original model representation (red), spectral sequencing (green), and our cache-oblivious metric (black). In the original layout, a large number of edges have edge spans greater than 600. Intuitively, our cache-oblivious metric favors edges that have small edge spans. Therefore, our layouts reduce cache misses.

changes in CMRF due to a local permutation are correspondingly small.

Monotonicity: We assume that the CMRF is a monotonically non-decreasing function of edge span. As we access vertices that are farther away from the current vertex (i.e. the edge spans increases), the probability of having a cache miss increases, until eventually leveling off at 1.

6.2.4 Cache-oblivious Metric

Our cache-oblivious metric is used to decide whether a local permutation decreases the expected number of cache misses, which due to the invariance of p_i is true if the following inequality holds:

$$\sum_{i=1}^{n-1}(|E_i| + \Delta|E_i|)p_i < \sum_{i=1}^{n-1}|E_i|p_i \Leftrightarrow \sum_{j=1}^{m}\Delta|E_{l(j)}|p_{l(j)} < 0 \tag{6.2}$$

Here $\Delta|E_i|$ is the signed change in the number of edges with edge span i after a local permutation and $n-1$ is maximum edge span for a mesh with n vertices. Furthermore, we let m denote the number of sets (among $E_1, E_2, \ldots, E_{n-1}$) whose cardinality changes because of the permutation, and let $l(j)$ denote the edge span associated with the jth such set, with $l(j) < l(j+1)$ and $m \ll n-1$.

Constant Edge Property: The total number of edges in a layout is the same before and after the local permutation. Hence

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| = 0 \tag{6.3}$$

Application of Monotonicity Assumption: Since we assume monotonicity on the CMRF, $p_{l(j)}$, we have the following relationship:

$$0 \leq p_1 \leq p_2 \leq \cdots \leq p_{m-1} \leq p_m \leq 1 \tag{6.4}$$

Inequality 6.2 given two constraints, Eq. (6.3) and Inequality (6.4), is our exact cache-oblivious metric.

6.2.5 Geometric Formulation

We reduce the computation of the expression in Eq. (6.2) to a geometric volume computation in an m dimensional hyperspace. Geometrically, the relationship represented in Eq. (6.4) defines a closed hyperspace in $\mathbb{R}^m$. We refer to this hyperspace as the *domain*. Eq. (6.2) defines a closed subspace within the domain of Eq. (6.4). Moreover, a dividing hyperplane defining this closed subspace passes through the point, $\{1, 1, \ldots, 1\} = P_O \in \mathbb{R}^m$, of the domain according to the constant edge property highlighted in Eq. (6.3). We also define the *top-polytope* of the domain as the polytope

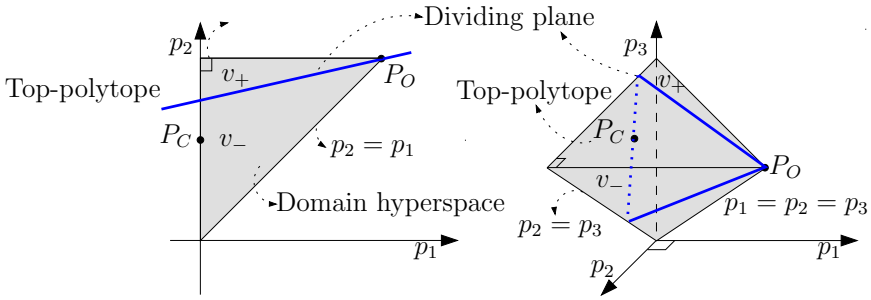


Figure 6.8: **Geometric volume computation:** The left figure shows a 2D geometric view of Eq. (6.2). The 3D version is shown in the right figure.

intersecting a hyperplane whose normal is parallel to an axis of p_1, with the closed hyperspace defined by Eq. (6.4). Moreover, we define V_+ to be the volume of the subspace represented in Eq. (6.2) and V_- to be the volume of its complement within the closed domain. These geometric concepts in 2 and 3 dimensions are illustrated in Fig. 6.8.

Volume Computation: Intuitively speaking, the volume V_+ corresponds to the set of cache configurations parameterized by $\{p_j\}$ for which we expect a reduction in cache misses. Since we assume all configurations to be equally likely, we probabilistically reduce the number of cache misses by accepting a local permutation whenever V_+ is larger than V_-.

Complexity of Volume Computation: The computation of the volume of a convex polytope defined by $m+1$ hyperplanes in m dimensions is a hard problem. The complexity of exact volume computation is $O(m^{m+1})$ (Lasserre & Zeron, 2001) and an approximate algorithm of complexity $O(m^5)$ is presented in (Kannan et al., 1997). In our application, each local permutation involves approximately 20–50 edges and these algorithms can be rather slow.

6.2.6 Fast and Approximate Metric

Given the complexity of exact volume computation, we use an approximate metric to check whether a local permutation would reduce the expected number of cache misses. In particular, we use a single sample point—the centroid of the top-polytope—as an estimate of $\{p_j\}$ to compute an approximate metric with low error.

Note that the dividing hyperplane between V_+ and V_- passes through the point P_O. Therefore, the ratio of V_+ to V_- is equal to the ratio of the $(m-1)$ dimensional areas formed by partitioning the top-polytope by the same dividing hyperplane. For example, in the 2D case, the result of volume comparison computed by substituting a centroid into Eq. (6.2) is exactly the same as the result of the 2D area comparison between V_+ and V_-. This formulation extends to 3D, but it introduces some error. The error is maximized when the dividing plane is parallel to one of the edges of the top-polytope and it is minimized (i.e., exactly zero) when the plane passes through one of its vertices.

We generalize this idea to m dimensions. P_C, the centroid of a top-polytope, is defined as $(\frac{0}{m}, \frac{1}{m}, \ldots, \frac{m-2}{m}, \frac{m-1}{m})$. By substituting P_C into Eq. 6.2 and canceling constants,

we have:

$$\sum_{j=1}^{m} \Delta |E_{l(j)}| j < 0 \tag{6.5}$$

If inequality (6.5) holds, we allow the local permutation. Based on this metric, we compute a layout, φ, that minimizes the number of cache misses.

Error Bounds on Approximate Metric: The approximate cache-oblivious metric has a worst case error of 26%, when the dividing hyperplane is parallel to one of the edges of the top-polytope. In practice, the worst case configuration is rare. In our benchmarks, we found that the actual error is typically much less (0.1-0.3%) than the

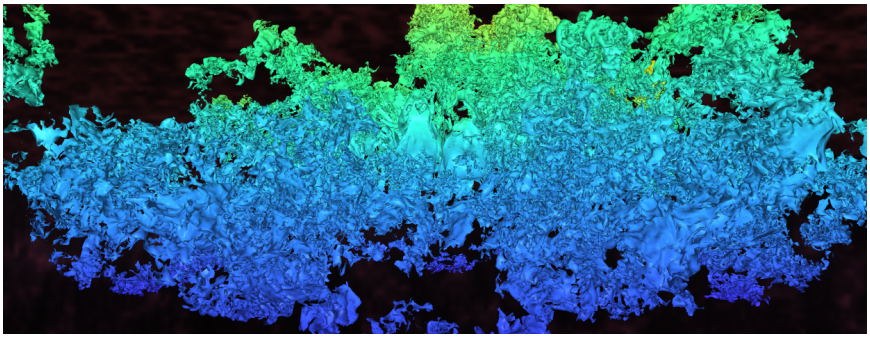


Figure 6.9: **Isosurface model:** This image shows a complex isosurface (100M triangles) generated from a 3D simulation of turbulent fluids mixing. Our layout reduces the vertex cache misses by more than a factor of four during view-dependent rendering. As a result, we improve the frame rate by 4 times as compared to prior approaches. We achieve a throughput of 90M tri/sec (at 30 fps) on a PC with an NVIDIA GeForce 6800 GPU.

worst case bound.

6.3 Layout Optimization

Given the cache-oblivious metric, our goal is to find the layout, φ, that minimizes the expected number of cache misses, defined in Eq. (6.1). This is a combinatorial optimization problem for graph layouts (Diaz et al., 2002). Finding a globally optimal layout is NP-hard (Garey et al., 1976) due to the large number of permutations of the set of vertices. Instead, we use a heuristic based on *multilevel minimization* that performs local permutations to compute a locally optimal layout.

6.3.1 Multilevel Minimization

Our multilevel algorithm consists of three main steps. First, a series of coarsening operations on the graph are computed. Next, we compute an ordering of vertices of the coarsest graph. Finally, we recursively expand the graph by reversing the coarsening operations and refine the ordering by performing *local permutations*. We will now

describe each of these steps in further detail.

Coarsening Step: The goal of the coarsening phase is to cluster vertices in order to reduce the size of the graph while preserving the essential properties needed to compute a good layout. We have tried two approaches: clustering via graph partitioning (Karypis & Kumar, 1998) and via streaming edge-collapse (Isenburg & Lindstrom, 2005), using only the topological structure of the graph as criterion for collapsing edges. As mentioned above, geometric locality can be preserved by adding additional edges to the graph between spatially close vertices.

Ordering Step: Given the coarsest graph of a handful of vertices, we list all possible orderings of its vertices and compute the costs based on the cache-oblivious metric from Eq. (6.5). We choose a vertex ordering that reduces the expected number of cache misses over all the other possible orderings.

Refinement Step: We reverse the sequence of coarsening operations applied earlier and exhaustively compute the locally optimal permutation of the subset of vertices involved in each corresponding refinement operation.

6.3.2 Local Permutation

We compute local permutations of the vertices during the ordering and refinement steps. A local permutation affects only a small number of vertices in the layout and changes the edge spans of those edges that are incident to these vertices. Therefore, we can efficiently evaluate the metric between two different vertex orderings. Each local permutation involves $k!$ possible orderings for k vertices, which during refinement replace their common parent in the evolving layout. For efficiency we restrict each coarsening operation to merge no more than $k = 5$ vertices at a time, and also limit

Model	Type	Vert. (M)	Tri. (M)	Size (MB)	Layout Comp. (min)
Dragon	s	0.4	0.8	33	0.25
Lucy	s	14.0	28.0	520	8
David	s	28.0	56.0	700	19
Double Eagle	c	77.7	81.7	3,346	56
Isosurface	i	50.5	100.0	2,543	49
Puget Sound	t	67.0	134.0	1,675	58
St. Matthew	s	186.0	372.0	9,611	176
Atlas	s	254.0	507.0	12,422	244

Table 6.1: **Layout Benchmarks:** Model complexity and time spent on layout computation are shown. Type indicates model type: s for scanned model, i for isosurface, c for CAD model, and t for terrain model. Vert. is the number of vertices and Tri. is the number of triangles of a model. Layout Comp. is time spent on layout computation.

the number of vertices in the coarsest graph to 5.

6.3.3 Out-of-Core Multilevel Optimization

The multilevel optimization algorithm needs to maintain an ordering of vertices along with a series of coarsening operations. For large meshes composed of hundreds of millions of vertices, it may not be possible to store all this information in main memory. In both of our graph partitioning and edge-collapse approaches, we compute a set of clusters, each containing a subset of vertices. Each cluster represents a subgraph and we compute an inter-cluster ordering among the clusters. We then follow the cluster ordering and compute a layout of all the vertices within each cluster using our multilevel minimization algorithm. Details of computing clusters in an out-of-core manner are explained in Section 4.2.1.

6.4 Implementation and Performance

In this section we describe our implementation and use cache coherent layouts to improve the performance of three applications: view-dependent rendering of massive mod-

els, collision detection between complex models, and isocontour extraction. Moreover, we used different kinds of models including CAD environments, scanned datasets, terrains, and isosurfaces to test the performance of cache coherent layouts. We also compare the performance of our metric with other metrics used for mesh layout.

6.4.1 Implementation

We have implemented our layout computation and out-of-core view-dependent rendering and collision detection algorithms on a 2.4GHz Pentium-4 PC with 1GB of RAM and a GeForce Ultra FX 6800 GPU with 256MB of video memory.

We use the METIS graph partitioning library (Karypis & Kumar, 1998) for coarsening operations to lay out vertex and bounding volume hierarchies. Our current unoptimized implementation of the out-of-core layout computation processes about 30K triangles per sec. In the case of the St. Matthew model, our second largest dataset, layout computation takes about 2.6 hours.

Memory-mapped I/O: Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files (Lindstrom & Pascucci, 2001). Windows XP can only map up to 2GB of user-addressable space. We overcome this limitation by mapping a small portion of the file at a time and remapping when data is required from outside this range.

Inducing a Layout: We compute only one of vertex and triangle layouts and induce the other layout rather than separately computing two layouts of the mesh in order to reduce layout computation time. First, we construct a vertex layout since the number of vertices is typically smaller than the number of triangles, hence, processing time of a vertex layout is smaller than that of a triangle layout. Then, as we access each vertex of the vertex layout, we sequentially store triangles incident on the vertex without any

Model	Double Eagle	Isosurface	St. Matthew
PoE	3	5	1
Frame rate	35	30	82
Rendering throughput(million tri./sec.)	47	90	106
Avg. Improvement	2.1	4.5	4.6
ACMR	1.58	0.75	0.72

Table 6.2: **View-Dependent Rendering** This table highlights the frame rate and rendering throughput for different models. We improve the rendering throughput and frame rates by $2.1 - 4.6$ times. The ACMR was computed with a buffer consisting of 24 vertices.

PoE	0.75	1	4	20
COL	0.71	0.72	0.73	0.74
SL	2.85	2.85	2.92	2.96

Table 6.3: **ACMR vs. PoE:** ACMRs are computed as we increase the PoE, i.e. use a more drastic simplification. The ACMRs of cache-oblivious layouts (COL) are still low even when a higher PoE is selected.

duplication in the triangle layout. We found that using the induced layouts at runtime cause a minor runtime performance loss–in our benchmark, less than 5%–compared to using layouts that are separately computed.

6.4.2 View-dependent rendering

View-dependent rendering and simplification are frequently used for interactive display of massive models. These algorithms precompute a multiresolution hierarchy of a large model (e.g. a vertex hierarchy). At runtime, a dynamic simplification of the model is computed by incrementally traversing the hierarchy until the desired pixels of error (PoE) tolerance in image space is met. Current view-dependent rendering algorithms are unable to achieve high polygon rendering throughput on the GPU for massive models composed of tens or hundreds of millions of triangles. It is not possible to compute rendering sequences at interactive rates for such massive models.

We use a clustered hierarchy of progressive meshes (CHPM) representation proposed

in Chapter 4 for view-dependent refinement along with occlusion culling and out-of-core data management. The CHPM-based refinement algorithm is very fast and most of the frame time is spent in rendering the simplified model. We precompute a cache-oblivious layout (COL) of the CHPM. The layout is used in order to reduce the cache misses for the vertex cache on the GPU. We computed layouts for three massive models including a CAD environment of a tanker with 127K separate objects (Fig. 6.3), a scanned model of St. Matthew (Fig. 6.2) and an isosurface model (Fig. 6.9). The details of these models are summarized in Table 6.1. We measured the performance of our algorithm along paths through the models.

Results

Table 6.2 highlights the benefit of COL over the simplification layout (SL), whose vertex layout and triangle layout are computed by the underlying simplification algorithm. We are able to increase the rendering throughput by a factor of of $2.1 - -4.6$ times by precomputing a COL of the CHPM of each model. We obtain a rendering throughput of 106M triangles per second on average, with a peak performance of 145M triangles per second.

Average Cache Miss Ratio (ACMR): The ACMR is defined by the ratio of the number of accessed vertices to the number of rendered triangles for a particular vertex cache size (Hoppe, 1999). If the number of triangles in the model is roughly twice the number of vertices (e.g. the St. Matthew and isosurface models), then the ACMR is within the interval $[0.5, 3]$. Therefore, the theoretical upper bound on cache miss reduction is a factor of 6. For a cache of 24 vertices, we improve the ACMR by a factor of 3.95 and get a 4.5 times improvement in the rendering throughput. On the other hand, if the number of vertices in the model is roughly the same as the number of triangles, as in the tanker model, then the ACMR is within the interval $[1, 3]$ and

the upper bound on cache miss reduction is 3 times. For this model, we improve the ACMR by a factor of 1.89 and the rendering throughput by a factor of 2.1. To verify the cache-oblivious nature of our layouts, we also simulated a FIFO vertex cache of configurable size and measured the ACMR as a function of cache size (Fig. 6.14). Table 6.3 shows the ACMR achieved by varying the PoE in the St. Matthew model.

Comparison with Other Layouts

We also compare our cache-oblivious layout with *universal rendering sequences* (URS) (Bogomjakov & Gotsman, 2002), Hoppe's rendering sequences (HRS) (Hoppe, 1999), and a Z-curve, which is a space filling curve. HRS is considered a cache-aware layout since it is optimized for a given cache size and replacement policy. On the other hand, Z-curve and URS are considered cache-oblivious layouts since they do not take advantage of any cache parameters.

Fig. 6.10 shows ACMRs of different rendering sequences on the Stanford bunny model. Since the number of triangles in the model is roughly twice the number of vertices, then the ACMR is within the interval $[0.5, 3]$. Moreover, optimal ACMR is $0.5 + O(\frac{1}{k})$ where k is the size of vertex cache (Bogomjakov & Gotsman, 2002). As you can see, ACMRs of our layout are very close to the optimal ACMRs and consistently shows superior performance over the universal rendering sequences among all the tested cache sizes. Although our layout shows less performance at cache size 16 compared to HRS, which is optimized for either cache size 12 or 16, our layout shows superior performance over HRS at cache size 8 and 64. These results are such because the fact that the cache-oblivious layout is not optimized at any particular cache sizes.

Fig. 6.11 shows ACMRs of COL and HRS as we decrease the resolution of the mesh at cache size 32. Since our layout is not optimized for any particular resolution of the mesh, our layout shows better performance over the HRS, which is optimized at the

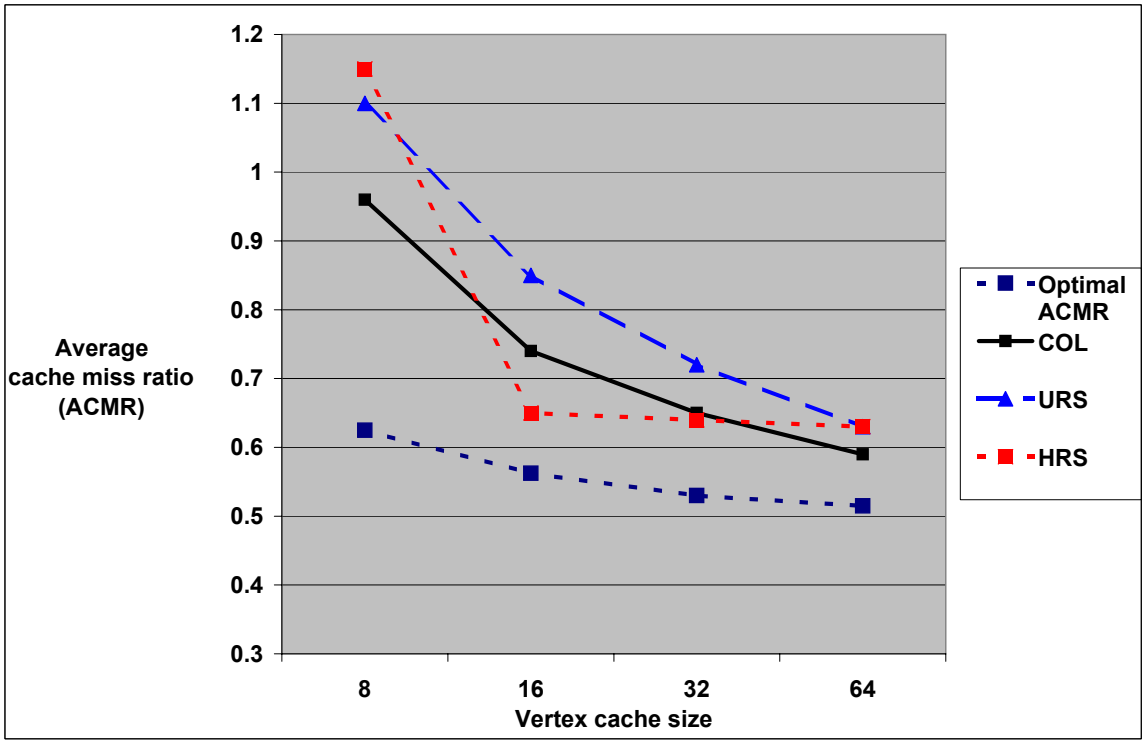


Figure 6.10: **Comparison with Other Rendering Sequences:** ACMRs of cache-oblivious layout (**COL**) are very close to optimal ACMRs. Also, COL consistently outperforms the universal rendering sequence (**URS**), and Hoppe's rendering sequence (**HRS**) at cache size 8 and 64; HRS is optimized at cache size 12 or 16.

finest resolution of the mesh.

Fig. 6.12 shows a comparison of ACMRs between our layout and the Z-curve on a power plant model, which has very irregular geometric distribution. Since any space filling curve including Z-curve assumes regular geometric distribution on the underlying models, the space filling curve may not demonstrate good performance on meshes that have irregular geometric distribution. As evidenced, our layout consistently shows better performance over the Z-curve. Moreover, the Z-curve shows even worse performance compared to original layout of the model.

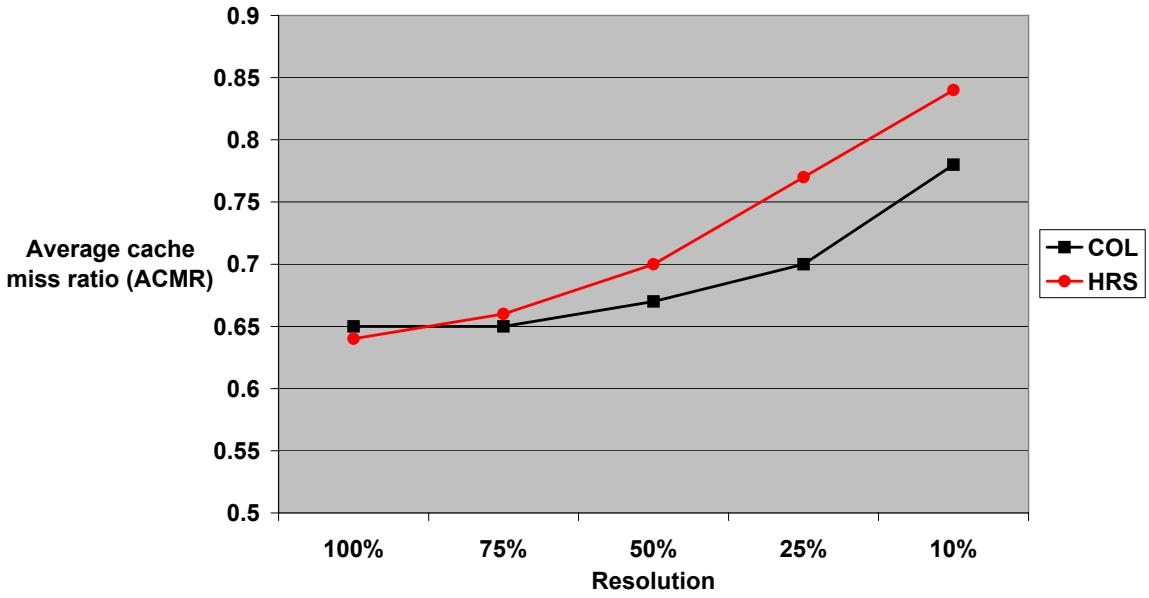


Figure 6.11: **ACMRs of Different Resolutions** ACMRs of cache-oblivious layout and Hoppe's rendering sequences are shown as the resolution of the mesh is decreasing at cache size 32.

6.4.3 Collision Detection

We use cache-oblivious layouts to improve the performance of collision detection algorithms based on bounding volume hierarchies. In particular, we compute layouts of OBB-trees (Gottschalk et al., 1996) and use them to accelerate collision queries within a dynamic simulator. Please refer to the Chapter 5 for more detail about collision detection algorithms based on bounding volume hierarchies.

We have tested the performance of our collision detection algorithm in a rigid body simulation, in which 20 dragons (800K triangles each) drop on the Lucy model (28M triangles). The details of these models are shown in Table 6.1. Fig. 6.13 shows a snapshot from our simulation.

We compared the performance of our cache-oblivious layout with the RAPID library (Gottschalk et al., 1996). The OBBs are precomputed and stored in memory-mapped files and only the ordering of the hierarchy is modified. We compared our cache-

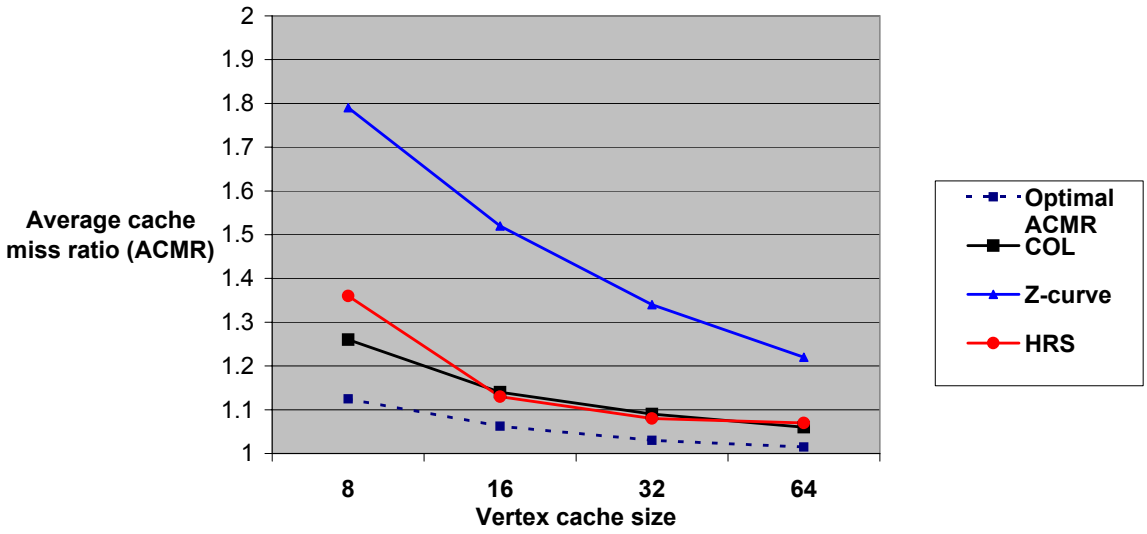


Figure 6.12: **Comparison with Space-Filling Curve:** Cache-oblivious layout (COL) consistently performs better performance than Z-curve on a power plant model, which has irregular geometric distribution.

oblivious layout with a depth-first layout (DFL) of OBB-trees. The DFL is computed by traversing the hierarchy from its root node in a depth-first order. We chose DFL because it preserves the spatial locality within the bounding volume hierarchy.

Results

On average, we are able to achieve twice improved performance over the depth-first layout on average. This is mainly due to reduced cache misses, including main memory page faults. We observe more than double the improvement whenever there are more broad contact regions. Such contacts trigger a higher number of page faults; in such situations we obtain a higher benefit from cache-oblivious layouts. The query times of collision detection during the dynamic simulation are shown in Fig. 6.15.

6.4.4 Isocontour Extraction

The problem of extracting an isocontour from an unstructured dataset frequently arises in geographic information systems and scientific visualization. We use an algorithm

Figure 6.13: **Dynamic Simulation:** Dragons consisting of 800K triangles are dropping on the Lucy model consisting of 28M triangles. We obtain twice improved performance by using COL on average.

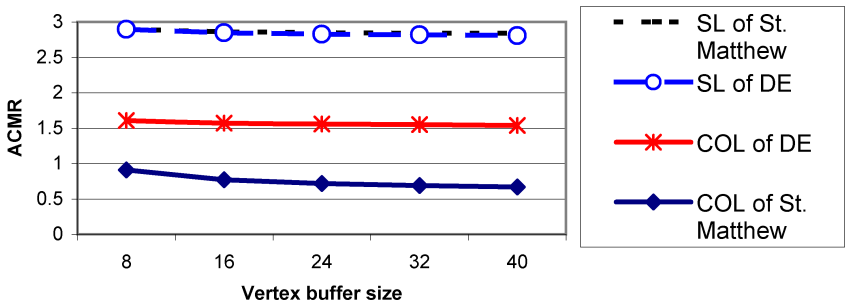


Figure 6.14: **ACMR vs. cache size:** ACMRs of cache-oblivious layout (COL) and simplification layout (SL) of the St. Matthew and double eagle tanker are shown. As the cache size increases, the improvement of COL becomes larger, and is 3.95 at a cache size of 24 in the St. Matthew model. Note that the lower bound on ACMR is 0.5 in St. Matthew and 1 in the double eagle tanker. The two SL curves almost overlap.

based on seeds sets (van Kreveld et al., 1997) to extract the isocontour of a single-resolution mesh. The running time of this algorithm is dominated by the traversal of the triangles intersecting the contour itself.

We use this algorithm to extract isocontours of a large terrain (Fig. 6.6) and compute equivalent geometric queries such as extracting ridge lines of a terrain[1] and cross-sections of large geometric models.

[1]For extracting a ridge line the seed point is a saddle and the propagation goes upward to the closest maxima instead of following an isocontour.

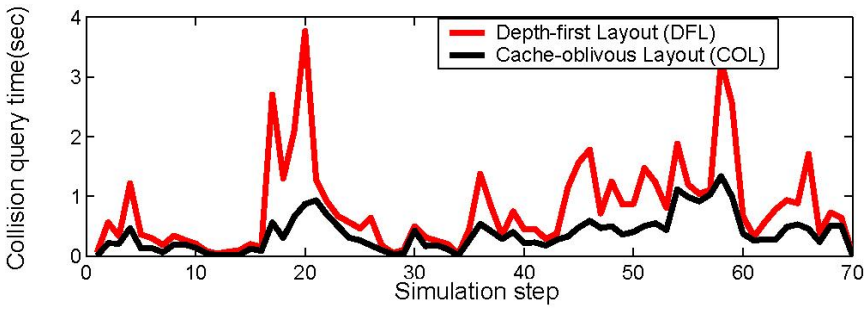


Figure 6.15: **Performance of Collision Detection:** Average query times for collision detection between the Lucy model and the dragon model with COL and DFL are shown. On average, we obtain 2 times improvement in the query time on average.

Comparison with other layouts

We compare the the performance of the isocontouring algorithm on four models, each stored in five different layouts. In addition to our cache-oblivious layout, we also store the meshes in geometric $X/Y/Z$ orders (vertices sorted by their position along the corresponding coordinate axis) and in spectral sequencing order (Diaz et al., 2002). We use edge-collapse as the coarsening step for computing cache-oblivious layouts and store all meshes in a streaming format (Isenburg & Lindstrom, 2005), which allows us to quickly compute the on-disk mesh data structure in a preprocess.

Table 6.4 reports the time in seconds required to compute an isocontour and a ridge line for the terrain models and to compute cross-sections of the other 3D models. The tests have been performed on a 1.3GHz Itanium Linux PC with 2GB of main memory. We take advantage of the 64-bit architecture and memory map the entire model. We do not perform any explicit paging. This way we ensure that the results are not biased to any particular layout.

The empirical data shows that our cache-oblivious layout minimizes the worst case cost of generic coherent traversals. The three layouts that are sorted by geometric direction along the X, Y, and Z axis show that the worst case performance is at least one order of magnitude slower than the best case, which is achieved by the layout that

Model	Puget Sound		Lucy	David	Atlas
Out. edg.	223K (Contour)	14K (Ridge)	17K (Section)	22K (Section)	38K (Section)
Cac. Obl.	026 (000.5)	003 (000.03)	03.3 (.04)	05.9 (.057)	010 (000.09)
Geom. X	232 (227.8)	001 (000.04)	01.2 (.04)	00.2 (.051)	015 (000.09)
Geom. Y	218 (215.5)	195 (185.10)	39.1 (.09)	60.7 (.103)	419 (379.78)
Geom. Z	011 (000.6)	135 (113.81)	26.1 (.09)	45.5 (.102)	443 (382.60)
Spec. Seq.	150 (127.3)	023 (000.04)	21.0 (.06)	43.1 (.068)	088 (000.10)

Table 6.4: **Isocontouring.** Time in seconds (on a 1.3GHz linux PC with 2GB of memory) for extracting an isocontour (or equivalent geometric queries) for several models stored each in five different mesh layouts: cache-oblivious, with vertices sorted by X/Y/Z geometric coordinate, and spectral sequencing. In parentheses we report the time for second immediate re-computation of the same contour when all the cache levels in the memory hierarchy have been loaded. In all cases, the performance of our cache-oblivious layout is comparable to the one optimized for the particular geometric query. This demonstrates the benefit of our layout for general applications.

happens to be perfectly aligned along the query direction. The spectral sequencing layout also does not perform well since the geometric query is unlikely to follow its streaming order. Our cache-oblivious layout consistently exhibits good performance.

The running times reported in parentheses in Table 6.4 are for a second immediate re-computation of the same contour, ridge line, or cross-section. They demonstrate the performance when all the cache levels have been loaded by the first computation. In this case our cache-oblivious layout is always as fast as the optimal case and can have a magnitude twice as fast as or even multiple times faster than the worst case. More importantly, this test demonstrates the cache-oblivious nature of the approach since performance advantages at different scales are achieved both when disk paging is necessary and when only internal memory and L2 caches are involved. In case of the Puget Sound terrain model, our cache-oblivious layout is the only layout that takes advantage of loaded cache levels for both the queries (i.e., isocontour and ridge line extraction).

6.5 Analysis and Limitations

We present a novel metric based on edge span distribution and CMRF to determine whether a local permutation on a layout reduces the expected number of cache misses. In practice, our algorithm computes layouts for which a high fraction of edges have very small edge spans. At the same time, a small number of edges in the layout can have a very large edge span, as shown in Fig. 6.7. This distribution of edge spans improves the performance because edges with small edge span increase the probability of a cache hit, while the actual length of very high-span edges has little impact on the likelihood of a cache hit.

Our multilevel minimization algorithm is efficient and produces reasonably good results for our applications. Moreover, our minimization algorithm maps very well to out-of-core computations and is able to handle very large graphs and meshes with hundreds of millions of triangles. We have applied our cache-oblivious layouts to models with irregular distribution of geometric primitives or irregular structures, for which prior algorithms based on space-filling curves may not work well.

Limitations: Our metric and layout computation algorithm has several limitations. The assumptions we make about invariance and monotonicity of CMRFs may not hold true for all applications, and our minimization algorithm does not necessarily compute a globally optimal solution. Our cache-oblivious layouts produce good improvements primarily in applications where the running time is dominated by data access.

Chapter 7

Cache-Oblivious Layouts of Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs) are frequently used to accelerate the performance of geometric processing and interactive graphics applications. These applications include ray tracing, visibility culling, collision detection, and geometric computations on massive datasets. Most of these algorithms precompute a BVH and traverse the hierarchy to speedup interference or proximity queries.

In the previous chapter, we used the cache-oblivious mesh layout algorithm (COML) to construct cache-oblivious layouts of BVHs. In order to apply the COML algorithm, an input graph that represents runtime access patterns is required. Each node of the graph is a bounding volume (BV) node of the BVH. An edge between two nodes exists if they are likely to be accessed sequentially at runtime. We identified two different localities during traversals of a BVH at runtime. To represent these localities in the input graph, we construct edges between two BV nodes that are likely to be accessed together due to the localities. However, we do not propose a unified way of assigning weight for created edges since two localities have their own different properties. Also, an output layout of BVH is drastically affected depending on the weight assignment method. In this chapter, we introduce a cache-oblivious layout algorithm that separately deals with two different localities during traversals of BVHs in order to avoid

this problem.

Main Results: We present a novel algorithm that computes cache-oblivious layouts of BVHs of large models. We make a very weak assumption of random, but cache-coherent access patterns on data access or traversal patterns of the hierarchy at runtime. Our algorithm is general and applicable to all kinds of BVHs that can be represented as a tree. Furthermore, our approach is cache-oblivious, in that it does not require any knowledge of cache parameters.

During traversal of a BVH, we identify two different localities: parent-child and spatial localities. Our algorithm considers these two localities separately in order to reduce the number of cache misses and the working set size. Firstly, we decompose the BVH into a set of clusters by considering parent-child locality to minimize the number of cache misses. Our cluster decomposition algorithm uses a probabilistic formulation based on tree packing (Gil & Itai, 1999). Secondly, we compute an ordering of clusters based on the cache-oblivious metric proposed in Section 6.2.6 by considering spatial locality between clusters. We recursively perform these two operations until all the BV nodes are ordered.

We use our algorithm to compute layouts of OBB trees of large models composed of 100K to millions of triangles. We use these layouts to perform collision queries based on BVHs of models. We compare their performance with prior algorithms and implementations and are able to achieve $2 - 5$ times performance improvement as compared to depth-first ordering of BVHs. Overall, our approach offers the following benefits:

1. **Generality:** Our algorithm is general and applicable to all kind of BVHs. It does not require any knowledge of cache parameters or block sizes of a memory hierarchy.

2. **Applicability:** Our algorithm does not require any modification of BVH-based algorithms or the runtime application. We simply compute cache-oblivious layouts of BVHs without making any assumptions about the applications.

3. **Improved performance:** Our layouts reduce the number of cache misses during traversals of BVHs. We are able to improve the performance of collision queries during dynamic simulation by $2-5$ times by using our layouts. Main improvements over the previous cache-oblivious mesh layout algorithms are due to a new layout algorithm and a probability computation method that can better capture runtime access patterns of proximity queries.

Organization: The rest of the chapter is organized in the following manner. We give an overview and two types localities during traversal of BVHs in Section 7.1. We present our greedy algorithm to compute layouts in Section 7.2 and describe its performance in Section 7.3. We compare our algorithm with prior approaches and discuss some of its limitations in Section 7.4.

7.1 Coherent Access Patterns on BVHs

In this section, we give an overview of BVHs and introduce two localities that are used to compute a cache-oblivious layout of a BVH. We also give a brief overview of prior work on cache-oblivious metric and packing trees, which are utilized by our layout computation algorithm. We also define some of the terminology used in the rest of the paper.

7.1.1 Interference and Proximity Queries

We use interference and proximity queries as a driving application to explain the concepts behind computing cache-oblivious layouts of BVHs. These algorithms take two

Figure 7.1: **Collision Detection:** We compute cache-oblivious layouts of bounding volume hierarchies (i.e. OBBTrees) of the Lucy (28M triangles) and 50 dragons (each 0.8M triangles). We use our cache-oblivious layouts to reduce the number of cache misses and to improve the performance of collision queries between the Lucy and 50 dragons. We achieve 5 times speed increase as compared to depth-first ordering of BVHs, without making any changes to the underlying collision algorithm.

objects: either two moving objects or one object and a ray as input. The runtime algorithm traverses the BVHs of each object using a depth-first order or breadth-first order. The depth-first order is used in cases when we need to check for ray-object intersection for ray-tracing or to check whether two objects collide. The breadth-first order is preferred when the runtime algorithm can be interrupted with approximate results at any time (e.g., constant frame-rate rendering of large models).

Extensive work has been done on evaluating the performance of different BVHs for ray-tracing and proximity queries. These include the cost equations for ray-tracing (Weghorst et al., 1984) and collision detection (Gottschalk et al., 1996; Klosowski et al., 1998). These cost equations take into account the tightness of fit for a BV and the relative cost of computing intersections or overlaps with those BVs based on the traversal pattern. However, these formulations do not take into account the cost of memory accesses or cache misses while traversing the BVHs. As the gap between

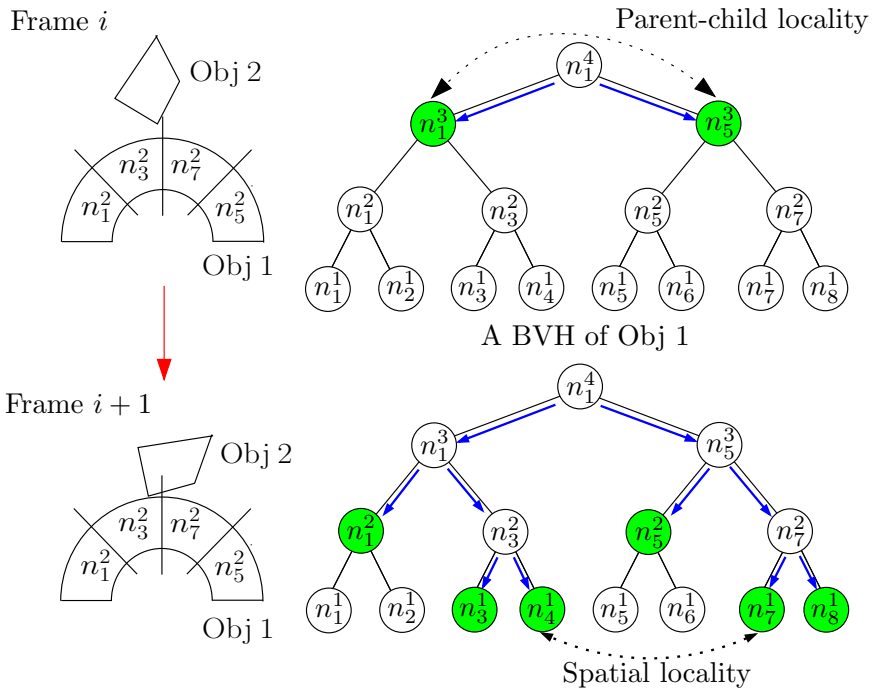


Figure 7.2: **Two localities within BVHs:** We show two successive frames from a dynamic simulation and the changes in access patterns (shown with blue arrows) of a BVH. In this simulation object 2 drops on object 1, as shown on the left. The access pattern of the BVH of object 1 during each frame is shown on the right. The BVs from the 2nd level in the BVH are shown within object 1 on the left to highlight the overlap tests performed between the BVs. We also illustrate the front traversed within each BVH during each frame in green. The top BVH shows the parent-child locality, when the root node, n_1^4, of the BVH of object 1 collides with the BVs of object 2. Once the root node is accessed, its two children are also accessed. During frame $i+1$, object 2 is colliding with object 1. In this configuration, the BVs n_3^2 and n_7^2 are accessed due to their spatial locality.

processor speed and memory access time widens, previous cost functions do not provide an accurate analysis of BVH-based algorithms for large models.

7.1.2 Layout of BVH

We use the following notation to represent the BVs of a BVH. We define $n_i = n_i^1$ as the ith BV node at the leaf level of the hierarchy and n_i^k as a BV node at the kth level of the hierarchy. We also define $Left(n_i^k)$ and $Right(n_i^k)$ to be the left and right child nodes of the n_i^k. They are represented as n_i^{k-1} and $n_{i+2^{k-2}}^{k-1}$, respectively. An example

of a BVH along with its leaf and intermediate level nodes is shown in Fig. 7.2.

A layout of a BVH is a linear sequence of BV nodes and triangles of the BVH. Triangles of the BVH are stored in the leaf nodes of the hierarchy. Formally speaking, a BVH is a directed acyclic graph, $G(N, A)$, where N is a set of BV nodes, n_i^k, of the BVH and A is a set of directed edges from a parent node, n_i^k, to each child node, $Left(n_i^k)$ and $Right(n_i^k)$, in the BVH. A layout of a BVH is comprised of two layouts: a BV layout and a triangle layout. A BV layout of a BVH, $G(N, A)$, is a one-to-one mapping of BVs to positions in the layout, $\varphi : N \rightarrow \{1, \ldots, |N|\}$. Our goal is to find a mapping, φ, that minimizes the number of cache misses and the size of the working set during the traversal of the BVH at runtime. Similarly, we also compute a triangle layout to minimize cache misses and the working set size during BVH traversals.

7.1.3 Access Patterns during BVH Traversal

Typical interference and proximity queries traverse BVHs as long as each query between two BVs reports close proximity between them. Our goal is to minimize the number of cache misses and the size of the working set during the traversal.

We decompose the access pattern during the traversal into a set of search queries. We define a *search query*, $S(n_i^k)$, to be the traversal from the root node of the BVH to the node, n_i^k, which can be either a leaf or an intermediate node of the BVH. Let us assume that the traversal starts from the root node and ends at nodes, $n_{i(1)}^{k(1)}, \ldots, n_{i(m)}^{k(m)}$ $(= BV_1, \ldots, BV_m)$. In this case, the nodes, $(BV_1, \ldots, BV_m)$, define a front of the BVH for this traversal. We represent this traversal as the union of m different search queries, $S(BV_j)$. An example of an access pattern between two colliding objects is shown in Fig. 7.2. In frame i, the collision query ends at n_1^3 and n_5^3 starting from the root node, n_1^4, of the BVH of object 1. We can represent the access patterns of this collision query with two search queries ending at n_1^3 and n_5^3.

There are two different localities that arise during the traversal: parent-child locality and spatial locality.

1. **Parent-child Locality:** Once a node of a hierarchy is accessed by a search query, it is likely that its child nodes would also be accessed soon. For example, in frame i of Fig. 7.2, if the root node of the BVH is accessed, its two child nodes, n_1^3 and n_5^3, are likely to be accessed soon in that frame. Moreover, after n_1^3 is accessed in frame i, its child nodes are likely to be accessed in the next frame.

2. **Spatial Locality:** Whenever a node is accessed by a search query, other nodes in close proximity are also highly likely to be accessed by other search queries. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is activated, other nearby nodes are either colliding or are in close proximity and may be accessed soon. In frame $i+1$ of Fig. 7.2, if one of two nodes, n_4^1 and n_7^1, is accessed, the other node is also likely to be accessed during that frame or subsequent frames.

We separately consider each of these two localities and use each of them to compute the layout of a BVH. In the remainder of this section, we briefly summarize several known results related to these localities.

7.1.4 Parent-Child Locality

We use several results presented by Gil and Itai (Gil & Itai, 1999) to compute a cache-coherent layout of a BVH. Gil and Itai (Gil & Itai, 1999) address the problem of computing a good layout for performing search queries on a tree. They define and use two different measures of cache-coherence of a layout of a tree. The two measure are:

1. **The number of cache misses:** $PF^1(BV_i)$ is defined as the number of cache misses, given a cache that can hold only single cache block during the traversal

of a search query ending at BV_i.

2. **The size of working set:** A working set during the traversal of the search query is a set of different cache blocks that are accessed. $WS(BV_i)$ is defined as the size of the working set.

Intuitively speaking, $PF^1(BV_i)$ is the number of times that accessing BVs cross boundaries of cache blocks of the layout during the traversal. Moreover, (Gil & Itai, 1999) define a probability function, $Pr(BV_i)$, that gives a measure of how many times BV_i is accessed during any search query on the tree. The expected size of working set, WS, of the layout can be formulated as:

$$WS = \sum_{BV_i \in \text{BVH}} Pr(BV_i) WS(BV_i), \tag{7.1}$$

for all nodes BV_i in the hierarchy. Similarly, we can define the expected number of cache misses, PF^1, of a layout by multiplying $Pr(BV_i)$ with $PF^1(BV_i)$ for all nodes BV_i in the tree.

Lemma 1 **(Convexity):** *If a layout of a tree is optimal given the metric PF^1 or WS, the layout is convex (Gil & Itai, 1999).*

The layout of a tree is convex if all the intermediate BVs between BV_0 and BV_k are stored in the same block when a node BV_0 and its descendant BV_k are stored in the same cache block.

Lemma 2 **(Equivalence):** *A layout of a tree is optimal given PF^1 metric if and only if the layout is optimal given WS metric (Gil & Itai, 1999).*

Lemma 3 **(NP-Completeness):** *Computing a layout of a tree that is a WS-optimal with a minimum storage is NP-Complete (Gil & Itai, 1999).*

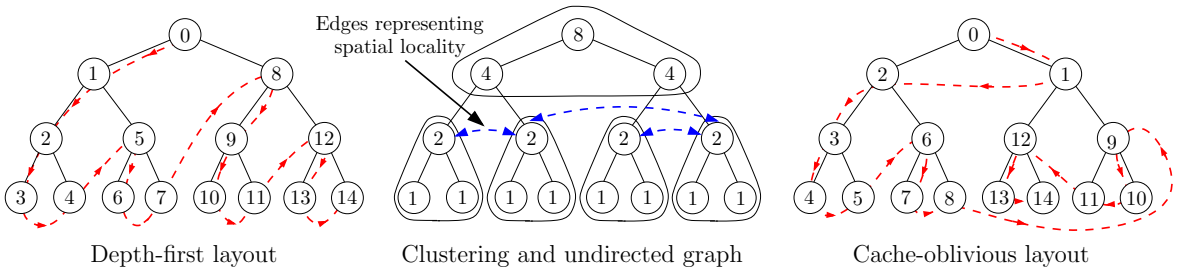


Figure 7.3: **Layout computation of a BVH:** A depth-first layout of a BVH is shown in the leftmost figure and a cache-oblivious layout of the same tree is shown in the rightmost figure. The number within each BV node in the leftmost and the rightmost figures is an index of the ordering of BVs in the layout. The middle figure shows the output of the clustering step. The topmost cluster is the root cluster and the rest are child clusters. Edges, shown in blue, are drawn between the root BVs of the child clusters that are nearby according to spatial relationships shown in Fig. 7.2. The number of each BV on the middle figure is an un-normalized probability assigned to each BV of the BVH.

We use these properties and lemmas to design our cache-oblivious layout algorithm that considers parent-child locality during the traversal of search queries.

7.1.5 Spatial Locality

We use the metric to construct cache-oblivious layouts of geometric meshes explained at Section 6.2.6. Depending on a spatial locality between two elements, we create an edge consisting of two vertices with a weight that is inversely proportional to the spatial locality; we use an inverse of distance between two elements as the weight.

In practice, this metric favors a local permutation that results in shorter edge spans considering weights of edges. However, the metric does not minimize the sum of edge spans like the minimum linear arrangement (MLA) (Diaz et al., 2002).

7.2 Layout Computation

We present a greedy algorithm to compute the cache-oblivious layout of a BVH. We take advantage of properties and lemmas explained in the previous section in order to

construct cache-coherent layouts of BVHs. Also, our algorithm consists of two main components, which separately consider parent-child and spatial localities.

7.2.1 Overall Algorithm

If we assume a particular cache size, we can compute how many nodes fit into a cache block. From this information, we can decompose an input BVH into a set of clusters, whose sizes are same to the size of a cache block. However, we do not assume any particular cache size and construct a layout that works well with any cache parameter. To achieve this goal, we recursively compute clusters; we first decompose an input BVH into a set of clusters and, then, recursively decompose each cluster in the same manner. Moreover, since cache block boundaries can lie anywhere within layouts that map to clusters, it is very important to have a cache-coherent ordering for the computed clusters at each level of recursion to further improve cache-coherence of the computed layout.

Our algorithm has two different components that separately handle parent-child and spatial localities. In particular, the first part of the algorithm decomposes a BVH into a set of clusters that minimize the cache misses for parent-child locality. The clusters are classified as a root cluster and child clusters. The root cluster contains the root node of the hierarchy and the child clusters are created for each child node whose parent node is a leaf node of the root cluster. Also, the second part of the algorithm computes an ordering of the clusters and stores the root cluster at the beginning of the ordering. The ordering of child clusters is computed by considering their spatial locality and relying on the cache-oblivious mesh layout algorithm described in Chapter 6. We recursively apply this two-fold procedure to compute an ordering of all the BVs in the BVH. An example of root and child clusters for a complete tree is shown in Fig.

7.3.

Cluster Decomposition: For each level of recursion, we decompose the BVH into a set of clusters that have approximately the same number of BV nodes belonging to each of the decomposed clusters. Suppose that a root cluster has B BV nodes. Then, the root cluster has $B+1$ child clusters; therefore, we decompose the input BVH into $B+2$ clusters. Assuming that each cluster is reasonably balanced in terms of the number of BV nodes belonging to each cluster, $B \times (B+2)$ should be bigger than n, which is the number of nodes in the BVH to contain all the nodes in the BVH. Therefore, B should be $\lceil \sqrt{n+1} \rceil$.

7.2.2 Cluster Computation

We partition the whole BVH into $B+2$ clusters, where B is the number of nodes in the root cluster and is set to $\lceil \sqrt{n+1} \rceil$.

We assign a probability, $Pr(n_i^k)$, to a BV, n_i^k, that the node is accessed during traversal of a search query on the BVH. In general, it is very hard to predict the probability at preprocessing time since we are unaware of the types of objects that will be used at runtime for collision detection between two objects.

We assume that each leaf BV node of the BVH has similar volume and, thus, is equally like to collide with a BV of another object at runtime. Therefore, $Pr(n_i^k)$ increases with the number leaf nodes that are in the sub-tree of the node, n_i^k. Therefore, $Pr(n_i^k)$ is formulated as the following equation:

$$Pr(n_i^k) = \begin{cases} 1 & \text{if BV is a leaf,} \\ Pr(Left(n_i^k)) + Pr(Right(n_i^k)) & \text{otherwise.} \end{cases}$$

We normalize the probability by dividing each $Pr(n_i^k)$ by the sum of probabilities of all the nodes in the hierarchy.

Our goal is to store BV nodes, which are accessed together due to the parent-child locality, into the same cluster in order to minimize the number of cache misses. The probability assigned to each node also can be considered as a probability that the node is accessed, given that a root node of a cluster is accessed due to the our probability computation method; we therefore achieve our goal by maximizing a sum of probabilities of BVs belonging to the root clusters. Moreover, maximizing the sum of probabilities is also strongly related to minimizing the expected size of working set. In other words, maximizing the sum of probabilities of BVs belonging to the root clusters also minimizes the probability to access nodes belonging to child clusters, in turn minimizes the number of times crossing boundaries of cache blocks of the layout, PF^1. Also, according to the Lemma 2, computing an optimal layout for PF^1 metric is again to compute an optimal layout that minimizes the expected size of working set, WS.

Since minimizing the working set and the number of cache misses for all possible search queries with a minimum space of a layout is NP-complete (as per Lemma 3), we employ a greedy algorithm to efficiently compute a cache-oblivious layout of a BVH. Our algorithm traverses the BVH and merges nodes from the root node of the BVH by maximizing the sum of the probabilities of the nodes in the root cluster. Once the root cluster has B nodes, we stop merging the nodes into the root cluster. Then, each child node of the leaf nodes in the root cluster consists of a child cluster containing all the nodes of its sub-tree. This process also maintains the convexity of the layout as defined by Lemma 1.

7.2.3 Layouts of Clusters

Given the computed clusters at each level of recursion, we compute a cache-oblivious ordering of the clusters by considering their spatial locality. During the recursions of

the overall algorithm, the number of BV nodes belonging to each cluster is roughly reduced by a factor of $B + 2$, achieved by performing the cluster computation at every recursion. This causes huge differences between sizes of clusters created during the previous level of the recursion and the current level of the recursion. Therefore, it is important to compute a cache-coherent ordering of clusters in order to further reduce the cache misses. This is because of high likelihood that the size of a cache block can lie between the cluster size in the previous level and current level of recursion.

We place the root cluster at the beginning of the ordering of clusters since the traversal typically starts from the root node of the BVH. In order to compute an ordering of child clusters, we construct an undirected graph where the child clusters are the nodes of the graph. We connect two clusters by assigning an edge in the graph if they are in close proximity. We define close proximity between child clusters by computing the K nearest neighbors for each node of the graph and use that information to compute the edges in the graph. An example of an undirected graph between child clusters is shown in the middle BVH of Fig. 7.3.

Once a graph is constructed, we compute a cache-oblivious layout from the graph that represents the access patterns between the child clusters. This is done in the same manner in which we computed the cache-oblivious mesh layout algorithm described in Chapter 6. An example of a cache-oblivious layout of a complete tree is shown in the rightmost figure of Fig. 7.3.

7.2.4 Triangle Layout

Once a set of BV pairs are computed during the traversal of BVHs of two objects, exact query computation based on triangles of leaf nodes is performed. We extract a triangle layout from a BV layout of the BVH for efficient layout computation. If we encounter leaf nodes of the hierarchy as we traverse the BV layout, we sequentially

order the triangles stored in the BVs. Since we perform overlap tests as sequentially following stored order of triangles belonging in a leaf node, any ordering of triangles in a leaf node does not have meaningful impact on the performance of triangle overlaps. We have observed that this simple algorithm, used to compute a triangle layout, works well in practice.

7.2.5 Out-of-Core Algorithm

Our goal is to compute BVH layouts of large meshes composed of millions of triangles. Our greedy layout algorithm needs to maintain an ordering of BVs along with the BVH. It may not be possible to store all this information in main memory. In order to avoid the high memory requirement, we employ three steps to compute a layout of a massive model. These steps are as follows:

- **Mesh decomposition:** We decompose the input mesh into portions of the mesh, each of which fits in main memory in order to perform the rest of steps in an in-core manner. This operation was previously explained in more detail in Section 4.2.1. We will call each portion of the mesh a leaf cluster. Then, we construct a BVH from the computed leaf clusters of the mesh. Please note that this BVH is very coarse since each leaf cluster contains several thousand triangles. We will call it a coarse BVH. We only keep the coarse BVH in the main memory and store decomposed geometry on disk. We also compute layout of the coarse BVH and save it on disk.

- **Processing leaf clusters:** We load geometry of each leaf cluster, compute a BVH for the geometry, and perform our layout computation algorithm for the BVH. We will call the BVH of the leaf cluster a leaf BVH. Since each leaf cluster is constructed such that it can fit into main memory, we can directly apply our

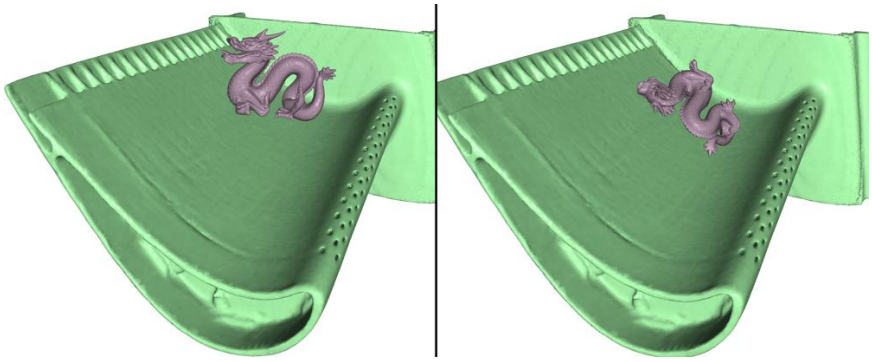


Figure 7.4: **Dynamic Simulation between Dragon and Turbine Models:** This image sequence shows discrete positions from our dynamic simulation between dragon and CAD turbine models. We are able to achieve more than 2 speedup by using our cache-oblivious layouts of BVHs of the models over depth-first ordering of BVHs.

layout computation algorithm without any additional disk IO operation. After computing a layout of the leaf BVH, we store the layout on disk and sequentially process other leaf clusters in the same manner.

- **Layout merging:** We already computed all the BVHs and their layouts, which are portions of a layout of the BVH of the input model. To construct one big layout of the BVH of the input model, we first read a layout of the coarse BVH and create a file with it. Then, we simply load the stored layout of each leaf BVH computed in the previous step and, then, add it to the file while appropriately stitching indices of the left and right child of each BV node.

7.3 Implementation and Performance

In this section we describe our implementation and highlight the performance of cache-oblivious layouts of BVHs of different polygonal models for collision detection during dynamic simulations.

Model	Triangles (M)	Size of BVH (MB)	Preprocessing time (min)
Bunny	0.06	13	0.06
Dragon	0.8	163	0.88
Turbine	1.7	331	2
Lucy	28	5,259	34

Table 7.1: **Benchmark Models:** Model complexity, sizes of BVHs, and preprocessing time to construct cache-oblivious layouts are shown.

7.3.1 Implementation

We have implemented our cache-oblivious layout algorithm and runtime collision detection on a 2.4GHz Pentium-IV PC, with 1GB of RAM. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files. Windows XP imposes a 2GB limitation for mapping a file to user-addressable address space. We overcome this limitation by mapping a 32MB portion of the file at a time and remapping when data is required from outside this range.

7.3.2 Benchmark Models

Our algorithm has been applied to different polygonal models. They include the Lucy model composed of more than 28 million polygons (Fig. 7.1), the CAD turbine model consisting of a single 1.7 million polygon object (Fig. 7.4), the dragon model consisting of 800K polygons, and the Stanford bunny model consisting of 67K polygons (Fig. 7.5). The details of these models are shown in Table 7.1.

7.3.3 Performance

We have applied our out-of-core algorithm to compute cache-oblivious layouts of BVHs of the models. Table. 7.1 presents preprocessing time for each model on the testing machine. An unoptimized implementation of our out-of-core algorithm can process 14K

triangle per second.

Collision Detection

We have tested our cache-oblivious layouts of BVHs of different models with collision detection during dynamic simulations. We have implemented an impulse based rigid body simulation (Mirtich & Canny, 1995) for dynamic simulation. We use OBBTrees (Gottschalk et al., 1996) for collision queries.

We compared the performance of our cache-oblivious layout of BVHs with the RAPID library (Gottschalk et al., 1996). The OBBs are precomputed into memory-mapped files and only the ordering of the hierarchy is modified. We compared our cache-oblivious layouts of BVHs (COLBVHs) with depth-first layouts (DFLs) of OBB-trees. The DFL is computed by traversing the hierarchy from its root node in a depth-first order. We chose DFL because it preserves the spatial locality within the bounding volume hierarchy. We also compared our COLBVH with a cache-oblivious layout based on a graph formulation (COML) by constructing the graph from OBB-trees. The COML, explained in Sec. 7.1.5, is computed by constructing an undirected graph. This is accomplished by generating edges between parent and child nodes and between nearby nodes on the same level of the BVH.

We have tested the performance of our collision detection algorithm in a rigid body simulation with three different benchmarks:

1. **Bunny and Dragon:** A bunny moves towards a dragon (Fig. 7.5).
2. **Dragon and Turbine:** A dragon drops onto the CAD turbine model and rests on it (Fig. 7.4).
3. **Dragons and Lucy:** 50 different dragons drop on the Lucy model (Fig. 7.1).

We are able to achieve $2-5$ times improvement in performance of collision queries

Models	Avg. collision detection time (sec)	Avg. number of overlap tests	Speedup over DFL	Speedup over COML
Bunny/Dragon	0.025	1900	2.7	1.7
Dragon/Turbine	0.073	6100	2.0	1.5
Lucy/Dragons	0.034	22,000	5	2.8

Table 7.2: **Runtime Performance of Collision Detection:** Here we see the average collision detection time and the average number of overlap tests during dynamic simulation between two models. Also the speedups over depth-first layouts (DFLs) and cache-oblivious mesh layouts (COMLs) are shown.

by using COLBVHs over DFLs in our benchmarks. Moreover, we are able to achieve $1.5 - 2.8$ times speed increase over COMLs on average in the same benchmarks. This improvement is mainly due to the improved clustering and more realistic probability computations. More detail comparison is in Sec. 7.4.1.

In Table 7.2, we report the average query times and the number of OBB overlap tests in each benchmark.

In the first and second benchmarks, we get more than twice increased speed as compared to the depth-first layouts. This is primarily due to the reduced cache misses including main memory page faults. We observe more than twice improvement whenever there are more broad contact regions. Such contacts trigger a higher number of page faults and in such situations we obtain a higher benefit from COLBVHs. Furthermore, the size of working sets of collision queries on COLBVHs is two to three times smaller than that of depth-first layouts. The query times of the first benchmark during the dynamic simulation are shown in Fig. 7.6.

In the third benchmark that consists of tens of millions of triangles, we get a 5 times speedup over the depth-first layouts. Since the size of BVHs is much larger than the size of main memory, the reduced working set size of cache-oblivious layouts of BVHs results in improved performance. The query times of the third benchmark are shown in Fig. 7.7.

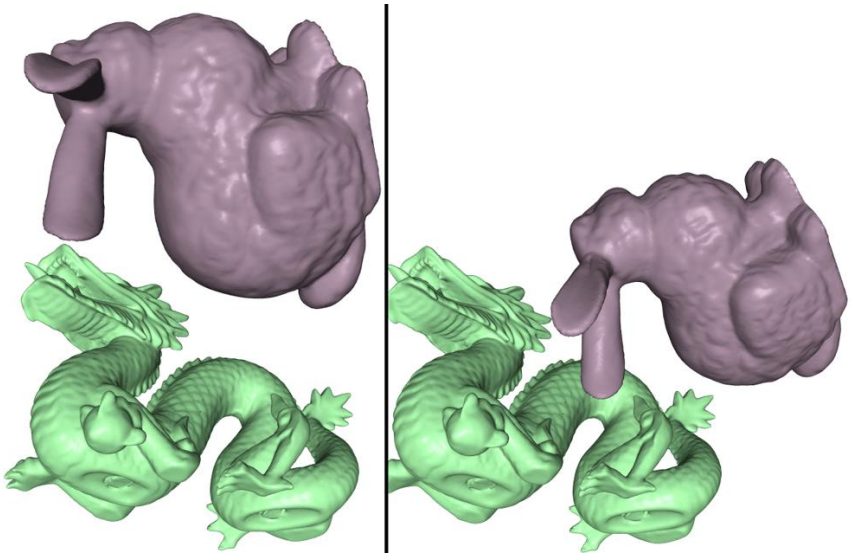


Figure 7.5: **Dynamic Simulation between Bunny and Dragon Models:** This image sequence shows discrete positions from our dynamic simulation between bunny and dragon models. We are able to achieve twice increased speeds by using our cache-oblivious layouts of BVHs of the models over depth-first ordering of BVHs.

7.4 Comparison and Limitations

In this section, we compare our algorithm with the cache-oblivious mesh layout described in Chapter 6 and discuss some of its limitations.

7.4.1 Comparison with Cache-Oblivious Mesh Layouts

We were able to achieve 1.5–2.8 times increased performance over the cache-oblivious mesh layout (COML). We attribute the following reasons to the improvement of our new algorithm:

- **Clustering method:** The COML method uses a graph partitioning during the multilevel minimization to compute cache-oblivious mesh layouts for any graph including a polygonal mesh or a BVH. However, there is no guarantee that clustering outputs of the graph partitioning on the input graph satisfy the convexity property. Therefore, the constructed layout of the BVH may be far from the opti-

mal layout that minimizes the size of the working set during traversal of proximity query. Instead, the layout algorithm optimized for BVHs always guarantees that clustering output satisfies the convexity property. Simultaneously, it maximizes the probabilities that BVs, which are accessed together due to the parent-child locality, are stored in a cluster.

- **Probability computation:** To construct an input graph for the COML method, edges should be created to represent access patterns of traversals of proximities queries. However, it is difficult to consistently compute weights of edges that represent parent-child or spatial localities in the graph. The edge creation methods for BVHs described in Chapter 6.1.3 do not represent adequately access patterns of the traversals. On the other hand, our algorithm (COLBVHs) works well for queries where the traversal can be represented as a set of search queries on BVHs and considers the two different localities separately.

7.4.2 Limitations

Our algorithm works well in our current set of benchmarks. However, it has certain limitations. Our greedy algorithm is based on several heuristics to compute cache-coherent layouts for parent-child locality. Therefore, there is no guarantee that our cache-oblivious layouts of BVHs would always reduce the number of cache misses. Moreover, our current layout algorithm assumes that traversals of proximity queries always starts from the root node of the BVH. However, some implementations of proximity queries may take advantage of temporal coherence and start from the collision nodes at previous frame in the current frame. Finally, our current probability computation assumes that leaf BVs have similar volume; therefore, the probability to access a node can be computed by counting how many leaf BVs are under the node. This works well with current data sets since there are almost regular shape for each triangle

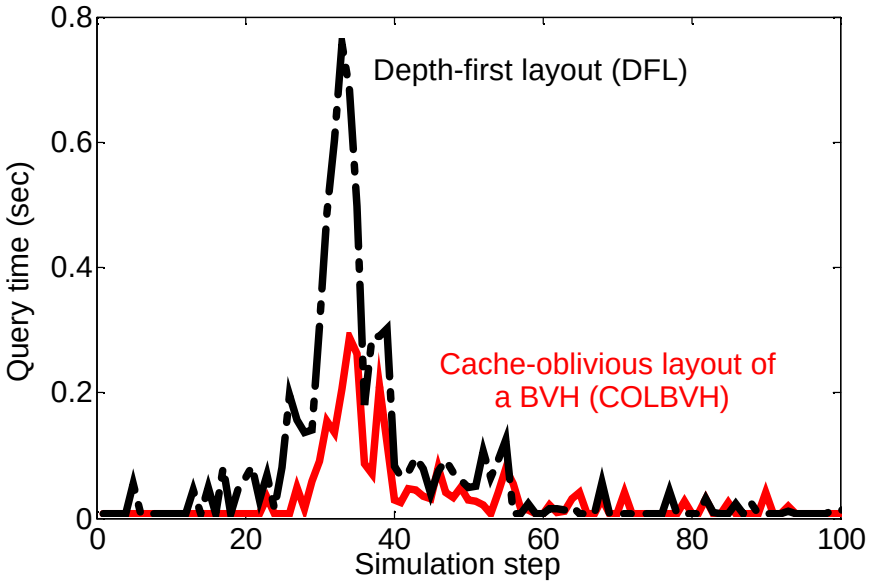


Figure 7.6: **Performance of Collision Detection:** Average query time for collision detection between the bunny and dragon models with COLBVH and DFL are shown. We obtain 2.7 times improvement in the query time.

and regular geometric distribution. We would like to extend our current probability computation for any kinds of polygonal models.

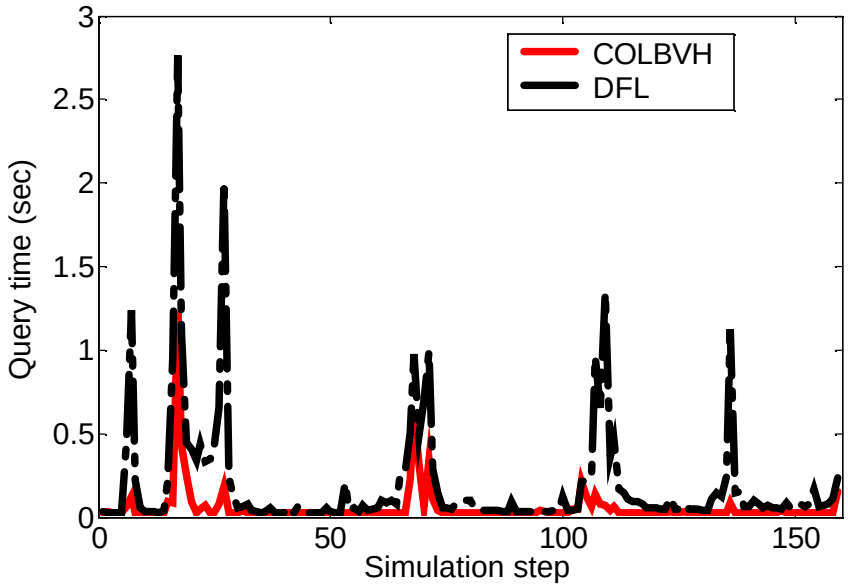


Figure 7.7: **Performance of Collision Detection:** Average query time for collision detection between the Lucy and 50 dragon models with COLBVH and DFL are shown. We obtain 5 times improvement in the query time over the DFL.

Chapter 8

Conclusion and Future Work

In this thesis we have proposed efficient dynamic simplification methods and cache-coherent layout algorithms and their applications on interactive view-dependent rendering and collision detection between massive and complex polygonal meshes consisting of tens or hundreds of millions of triangles.

Overall our algorithms have the following benefits:

- **Interactive performance:** Our dynamic simplification algorithms provide the capability to perform interactive view-dependent rendering and collision detection between massive and complex polygonal meshes. Moreover, our cache-coherent layout algorithms have improved cache utilization of applications without modifying runtime algorithms and applications.

- **Generality:** Our algorithms do not require any knowledge regarding input polygonal meshes. Therefore, we have been able to demonstrate our algorithms on a wide variety of polygonal meshes including scanned models, isosurfaces, and CAD models. Moreover, our algorithm can even handle polygon soups.

- **Applicable to massive models on commodity hardware:** Our algorithms have been successfully tested on massive models consisting of tens or hundreds of millions of triangles that cannot fit into main memory of commodity hardware.

- **High quality and accuracy:** By using dynamic simplification, we have minimized error caused by switching between different LODs. We have also quantified errors within bounds for both view-dependent rendering and contact-dependent collision detection.

In the following sections, we summarize our algorithms and highlight possible directions for future research.

8.1 Interactive Visualization

We have presented novel dynamic simplification algorithms for interactive out-of-core view-dependent rendering of complex and massive models. We propose the CHPM representation as a scene representation for efficient dynamic simplification. The CHPM allows us to perform coarse-grained as well as fine-grained refinement. It significantly reduces the refinement cost as compared to earlier approaches based on vertex hierarchies. The cluster hierarchy enables occlusion culling and out-of-core rendering. We also presented an out-of-core algorithm to compute CHPMs, which facilitate an integration of view-dependent simplification, occlusion culling, and out-of-core rendering. We have tested our algorithms on massive models with a few hundred million triangles and can render them at interactive rates using commodity graphics systems.

Future Work: In addition to addressing the limitations of our current approaches, we would like to further investigate the following three major research directions;

- **Achieving end-to-end interactivity:** Our algorithm precompute the scene representation (e.g., CHPM) for high-quality and interactive performance of massive models at runtime. Although we achieve interactive performance at runtime, it takes several hours to precompute the scene representation; for example, it

takes 10 hours to precompute a CHPM of St. Matthew model consisting of 372 million triangles. Ideally, we would like to achieve end-to-end interactivity. This would provide us with the ability to explore various data sets while eliminating excess waiting time.

- **Handling time-varying geometry:** Our algorithms efficiently compute dynamic simplification representation for static models. However, it is unclear how to extend current algorithms to handle time-varying geometry including animation models and scientific simulation data. One possible way to handle time-varying geometry is to consider time dimension as a fourth dimension during the process of simplifying the input 4D data sets. Moreover, memory efficient representation of time-varying geometry should be explored since the memory requirement for time-varying geometry is much higher than static data sets.

- **Robust out-of-core visibility algorithm:** Our current visibility techniques require high temporal coherence. If there is little temporal coherence, our visibility techniques waste considerable frame time to render an occlusion map that may not cull anything from the new viewpoint. Also, we may load too many clusters from geometry due to a poor quality of occlusion map. To address these issues, we would like to further investigate a robust out-of-core visibility algorithm that does not require high temporal coherence.

8.2 Approximate Collision Detection

We have presented a new algorithm for out-of-core collision detection using the CHPM representation. The algorithm has many benefits, which include:

- We are able to accelerate the computation using LODs while ensuring all contact regions are detected.

- Our algorithm efficiently handles models with tens of millions of triangles using out-of-core computations.

- The CHPM representation and supporting algorithms can handle models with arbitrary topology and polygon soups.

- We use a unified representation for collision detection and interactive rendering of massive models that uses a finite-memory footprint.

Future Work: There are several areas for future work. They can be classified as follows:

- **Handling dynamically deforming models:** There has been increasing attention regarding handling dynamically deforming models in computer graphics and computer games. This creates many challenges for interactive collision detections. Our current approach consisted of precomputing a multiresolution hierarchy to support interactive collision detection between massive models at runtime. Once a model is dynamically deforming (e.g., cloth), the multiresolution hierarchy needs to be updated to reflect the new geometry and topology of the model. We would like to investigate different representations to support efficient runtime updates of the model's multiresolution.

- **Other proximity queries and applications:** The current algorithm is designed primarily for collision detection. We would like to extend our algorithms to perform other proximity queries such as computing separation distance and penetration depth. Also, we would like to apply our LOD-based collision detection framework to several applications including motion planning, navigation, and dynamic simulation.

8.3 Cache-Oblivious Layouts

We have presented a novel approach to computing cache-oblivious layouts of large meshes and hierarchies including bounding volume hierarchies. We only make an assumption that the runtime applications has random, but cache-coherent access patterns and compute an ordering that results in high locality. We demonstrate that our formulation can be extended to computations of layouts of bounding volume and multiresolution hierarchies of large meshes. We use a probabilistic model to minimize the number of cache misses. Our preliminary results indicate that our metric succeeds in practice for reducing cache misses. Furthermore, we compute cache-oblivious layouts of different kinds of geometric datasets including scanned models, isosurfaces, terrain, and CAD environments with irregular distributions of primitives. We used our layouts to improve the performance of view-dependent rendering, collision detection and isocontour extraction by $2 - 20$ times without any modification of the algorithm or runtime applications.

In addition to the general layout algorithm, we have proposed a specialized layout algorithm for bounding volume hierarchies. We decompose the access patterns during a traversal into a union of a set of search queries and utilize parent-child and spatial localities between search queries. Our algorithm computes cache-coherent layouts by separately considering the two localities in a cache-oblivious manner. Furthermore, we applied our cache-oblivious layouts of BVHs to collision detection between complex models. We were able to achieve $2 - 5$ times improvements on the performance over depth-first layouts.

Future Work: There are various avenues for future work. We would like to address the limitations of our approaches and further investigate the following directions;

- **Application-dependent layout algorithms:** Our layout algorithms, which are based on graph-formulations, are application-independent; if a graph representing the access patterns of runtime application is provided, a cache-coherent layout of the graph can be automatically constructed. However, generally it is difficult to correctly capture the runtime access patterns and create the input graph for layout computation. We would like to explore automatic graph constructions method given runtime applications by using profiling method (Rubin et al., 2002).

- **Cache-aware layout algorithms:** We have only considered cache-oblivious layout algorithms. However, we infer that we can obtain further performance improvement by taking into account cache parameters such as block and cache size, in order to design an improved metric.

- **Optimality:** We would like to investigate how our layout is close to optimal performance of a layout of a polygonal mesh. We have conjectured from various experiments that performance of cache-oblivious layout is close to optimal. For example, the average cache miss ratio of our layout for view-dependent rendering is 30-100% more than the optimal cache miss ratio. By relying on theoretical analysis rather than empirical methods, we would like to more rigorously verify our immature conjecture.

- **Supporting multiresolution based on a 1D layout:** Pascucci and Frank (Pascucci & Frank, 2001) organized the layout of a volumetric grid such that the layout efficiently supports various geometric operations such as slicing and multiresolution rendering. Similarly, we would like to investigate the layout computation algorithm of an input mesh to support multiresolution rendering.

- **Handling time-varying data sets:** Our current method only deals with static data sets. We would like to extend the current method, allowing it to handle

various types of time-varying data sets such as animation data and scientific simulation data.

- **Application to other problems and other data sets:** We would like to apply our layout to improve the performance of algorithms for processing and manipulation of large meshes and bounding volume hierarchies in various applications including simplification, compression, smoothing, isosurface extraction, shadow generation (Govindaraju et al., 2003a; Lloyd et al., 2005), approximate collision detection proposed in Chapter 5, ray-tracing and, other fundamental graph algorithms like the shortest path algorithm. Also we would like to use our graph-based formulation to compute cache-coherent layouts for other kinds of datasets, including point primitives and unstructured volumetric grids.

Bibliography

Airey, J., Rohlf, J., & Brooks, F. (1990). Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50.

Aliaga, D., Cohen, J., Wilson, A., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Baker, E., Bastos, R., Whitton, M., Brooks, F., & Manocha, D. (1999). MMR: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 199–206.

Alstrup, S., Bende, M. A., Farach-Colton, E. D. D. . M., Rauhe, T., & Thorup, M. (2003). Efficient tree layout in a multilevel memory hierarchy. *Computing Research Repository (CoRR)*.

Arge, L., Brodal, G., & Fagerberg, R. (2004). Cache oblivious data structures. *Handbook on Data Structures and Applications*.

Arya, S. & Mount, D. M. (1993). Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280.

Bartholdi, J. & Goldsman, P. (2004). Multiresolution indexing of triangulated irregular networks. In *IEEE Transaction on Visualization and Computer Graphics*, pages 484–495.

Bartz, D., Meibner, M., & Huttner, T. (1999). Opengl assisted occlusion culling for large polygonal models. *Computer and Graphics*, 23(3):667–679.

Baxter, B., Sud, A., Govindaraju, N., & Manocha, D. (2002). GigaWalk: Interactive walkthrough of complex 3D environments. *Proc. of Eurographics Workshop on Rendering*, pages 203–214.

Beckmann, N., Kriegel, H., Schneider, R., & Seeger, B. (1990). The r*-tree: An efficient and robust access method for points and rectangles. *Proc. SIGMOD Conf. on Management of Data*, pages 322–331.

Bogomjakov, A. & Gotsman, C. (2002). Universal rendering sequences for transparent vertex caching of progressive meshes. In *Computer Graphics Forum*, pages 137–148.

Borgeat, L., Godin, G., Blais, F., Massicotte, P., & Lahanier, C. (2005). Gold: Interactive display of huge colored and textured models. *ACM SIGGRAPH*.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., & Scopigno, R. (2004). Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. volume 23, New York, NY, USA. ACM Press.

Cignoni, P., Montani, C., Rocchini, C., & Scopigno, R. (2003). External memory management and simplification of huge meshes. In *IEEE Transaction on Visualization and Computer Graphics*, pages 525–537.

Clark, J. H. (1976). Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19:547–554.

Cohen, J., Olano, M., & Manocha, D. (1998). Appearance preserving simplification. In *Proc. of ACM SIGGRAPH*, pages 115–122.

Cohen-Or, D., Chrysanthou, Y., Silva, C., & Durand, F. (2003). A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*.

Coleman, S. & McKinley, K. (1995). Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290.

Coorg, S. & Teller, S. (1997). Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

Correa, W., Klosowski, J., & Silva, C. (2002). iwalk: Interactive out-of-core rendering of large models. In *Technical Report TR-653-02, Princeton University*.

Corrêa, W. T. (2004). *New Techniques for Out-Of-Core Visualization of Large Datasets*. PhD thesis, Princeton University.

Corrêa, W. T., Klosowski, J. T., & Silva, C. T. (2003). Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, pages 1–8.

Cox, M. & Ellsworth, D. (1997). Application-controlled demand paging for out-of-core visualization. In *Proc. of IEEE Visualization*, pages 235–244.

Dachsbacher, C., Vogelgsang, C., & Stamminger, M. (2003). Sequential point trees. In *Proc. of ACM SIGGRAPH*.

DeCoro, C. & Pajarola, R. (2002). Xfastmesh: View-dependent meshing from external memory. In *IEEE Visualization*.

Deering, M. F. (1995). Geometry compression. In *ACM SIGGRAPH*, pages 13–20.

Diaz, J., Petit, J., & Serna, M. (2002). A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356.

Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., & Mineev-Weinstein, M. B. (1997). ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proc. IEEE Visualization*, pages 81–88.

El-Sana, J., Azanli, E., & Varshney, A. (1999). Skip strips: maintaining triangle strips for view-dependent rendering. *IEEE Visualization*.

El-Sana, J. & Bachmat, E. (2002). Optimized view-dependent rendering for large polygonal dataset. *IEEE Visualization*, pages 77–84.

El-Sana, J. & Chiang, Y.-J. (2000). External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150.

El-Sana, J., Sokolovsky, N., & Silva, C. (2001). Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*.

El-Sana, J. & Varshney, A. (1999). Generalized view-dependent simplification. *Computer Graphics Forum*, pages C83–C94.

Erikson, C. & Manocha, D. (1999). GAPS: General and automatic polygon simplification. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

Erikson, C., Manocha, D., & Baxter, B. (2001). Hlods for fast display of large static and dynamic environments. *Proc. of ACM Symposium on Interactive 3D Graphics*.

Fishburn, P., Tetali, P., & Winkler, P. (2000). Optimal linear arrangement of a rectangular grid. *Discrete Mathematics*, 213(1):123–139.

Floriani, L. D., Magillo, P., & Puppo, E. (1997). Building and traversing a surface at variable resolution. In *IEEE Visualization*.

Floriani, L. D., Magillo, P., & Puppo, E. (1998). Efficient implementation of multi-triangulations. In *IEEE Visualization*.

Franquesa-Niubo, M. & Brunet, P. (2003). Collision prediction using mktrees. *Proc. CEIG*, pages 217–232.

Frigo, M., Leiserson, C., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In *Foundations of Computer Science*, pages 285–297.

Funkhouser, T., Khorramabadi, D., Sequin, C., & Teller, S. (1996). The ucb system for interactive visualization of large architectural models. *Presence*, 5(1):13–44.

Funkhouser, T. A. & Squin, C. H. (1993). Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254.

Garey, M., Johnson, D., & Stockmeyer, L. (1976). Some simplified np-complete graph problems. *Theoretical Computer Science 1*, pages 237–267.

Garland, M. & Heckbert, P. (1997). Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216.

Garland, M., Willmott, A., & Heckbert, P. (2001). Hierarchical face clustering on polygonal surfaces. In *Proc. of 2001 Symposium on Interactive 3D Graphics*, pages 49–58.

Gil, J. & Itai, A. (1999). How to pack trees. *Journal of Algorithms*.

Gobbetti, E. & Marton, F. (2005). Far Voxels – a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics*, 24(3):878–885.

Gopi, M. & Eppstein, D. (2004). Single-strip triangulation of manifolds with arbitrary topology. In *EUROGRAPHICS*, pages 371–379.

Gottschalk, S., Lin, M., & Manocha, D. (1996). OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180.

Govindaraju, N., Lin, M., & Manocha, D. (2004). Fast and reliable collision detection using graphics hardware. *Proc. of ACM VRST*.

Govindaraju, N., Lloyd, B., Yoon, S., Sud, A., & Manocha, D. (2003a). Interactive shadow generation in complex environments. *Proc. of ACM SIGGRAPH/ACM Trans. on Graphics*, 22(3):501–510.

Govindaraju, N., Redon, S., Lin, M., & Manocha, D. (2003b). CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32.

Govindaraju, N., Sud, A., Yoon, S., & Manocha, D. (2003c). Interactive visibility culling in complex environments with occlusion-switches. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 103–112.

Greene, N. (2001). Occlusion culling with optimized hierarchical z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY, # 30*.

Greene, N., Kass, M., & Miller, G. (1993). Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238.

Heber, G., Biswas, R., & Gao, G. (2000). Self-avoiding walks over adaptive unstructured grids. In *Concurrency: Practice and Experience*, pages 85–109.

Heidelberger, B., Teschner, M., & Gross, M. (2003). Real-time volumetric intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*, pages 461–468.

Hendrickson, B. & Leland, R. (1995). A multilevel algorithm for partitioning graphs. In *Super Computing*.

Hillesland, K., Salomon, B., Lastra, A., & Manocha, D. (2002). Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, University of North Carolina.

Hoppe, H. (1996). Progressive meshes. In *Proc. of ACM SIGGRAPH*, pages 99–108.

Hoppe, H. (1997). View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, pages 189–198.

Hoppe, H. (1998). Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization Conference Proceedings*, pages 35–42.

Hoppe, H. (1999). Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, pages 269–276.

Hubbard, P. M. (1993). Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*.

Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15(3):179–210.

Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., & Zhang, H. (1997). Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, pages 1–10.

Isenburg, M. & Gumhold, S. (2003). Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, volume 22, pages 935–942.

Isenburg, M. & Lindstrom, P. (2005). Streaming meshes. *IEEE Visualization*, pages 231–238.

Isenburg, M., Lindstrom, P., Gumhold, S., & Snoeyink, J. (2003). Large mesh simplification using processing sequences. *IEEE Visualization*, pages 465–472.

Jimenez, P., Thomas, F., & Torras, C. (2001). 3d collision detection: A survey. *Computers and Graphics*, 25(2):269–285.

Jolliffe, I. (1986). Principle component analysis. In *Springer-Veriag*.

Juvan, M. & Mohar, B. (1992). Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics*, 36(2):153–168.

Kannan, R., Lovasz, L., & Simonovits, M. (1997). Random walks and an $O(n^5)$ time algorithm for volumes of convex sets. *Random Structures and Algorithms*, pages 1–50.

Karni, Z., Bogomjakov, A., & Gotsman, C. (2002). Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, pages 347–354.

Karypis, G. & Kumar, V. (1998). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, pages 96–129.

Kim, Y. J., Lin, M. C., & Manocha, D. (2002). Fast penetration depth computation using rasterization hardware and hierarchical refinement. *Proc. of Workshop on Algorithmic Foundations of Robotics*.

Klosowski, J., Held, M., Mitchell, J., Sowizral, H., & Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37.

Klosowski, J. & Silva, C. (2000). The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. on Visualization and Computer Graphics*, 6(2):108–123.

Klosowski, J. & Silva, C. (2001). Efficient conservative visiblity culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):365–379.

Knott, D. & Pai, D. K. (2003). CInDeR: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, pages 73–80.

Kobbelt, L., Campagna, S., & Seidel, H. (1998). General framework for mesh decimation. *Proc. Graphics Interface*.

Larsen, E., Gottschalk, S., Lin, M., & Manocha, D. (2000). Distance queries with rectangular swept sphere volumes. *Proc. of IEEE Int. Conference on Robotics and Automation*, pages 3719–3726.

Lasserre, J. B. & Zeron, E. S. (2001). A laplace transform algorithm for the volume of a convex polytope. *Journal of the ACM*, pages 1126–1140.

Lin, M. & Manocha, D. (2003). Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.

Lindstrom, P. (2000). Out-of-core simplification of large polygonal models. In *Proc. of ACM SIGGRAPH*.

Lindstrom, P. (2003). Out-of-core construction and visualization of multiresolution surfaces. In *ACM Symposium on Interactive 3D Graphics*.

Lindstrom, P., Koller, D., Ribarsky, W., Hughes, L. F., Faust, N., & Turner, G. (1996). Real-Time, continuous level of detail rendering of height fields. In Rushmeier, H., editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley. held in New Orleans, Louisiana, 04-09 August 1996.

Lindstrom, P. & Pascucci, V. (2001). Visualization of large terrains made easy. *IEEE Visualization*, pages 363–370.

Lindstrom, P. & Pascucci, V. (2002). Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. In *IEEE Transaction on Visualization and Computer Graphics*, pages 239–254.

Lindstrom, P. & Silva, C. (2001). A memory insensitive technique for large model simplification. In *Proc. of IEEE Visualization*, pages 121–126.

Lloyd, B., Yoon, S.-E., Tuft, D., & Manocha, D. (2005). Subdivided shadow maps. Technical report, University of North Carolina-Chapel Hill.

Luebke, D. & Erikson, C. (1997). View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*.

Luebke, D. & Georges, C. (1995). Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA.

Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., & Huebner, R. (2002). *Level of Detail for 3D Graphics*. Morgan-Kaufmann.

Meissner, M., Bartz, D., Huttner, T., Muller, G., & Einighammer, J. (2002). Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. *Computer and Graphics*.

Mirtich, B. & Canny, J. (1995). Impulse-based simulation of rigid bodies. In *Proc. of ACM Interactive 3D Graphics*, Monterey, CA.

Oliker, L., Li, X., Husbands, P., & Biswas, R. (2002). Effects of ordering strategies and programming paradigms on sparse matrix computations. In *SIAM Review*, pages 373–393.

O'Sullivan, C. & Dingliana, J. (2001). Collisions and perception. *ACM Trans. on Graphics*, 20(3):pp. 151–168.

Otaduy, M. A. & Lin, M. C. (2003). CLODs: Dual hierarchies for multiresolution collision detection. *Eurographics Symposium on Geometry Processing*, pages 94–101.

Pajarola, R. (2001). Fastmesh: Efficient view-dependent mesh. In *Proc. of Pacific Graphics*, pages 22–30.

Pascucci, V. & Frank, R. J. (2001). Global static indexing for real-time exploration of very large regular grids. In *Supercomputing*.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Kimberly, Keaton, ChristoforosKazyrakis, Thomas, R., & Yellick, K. (1997). A case for intelligent ram. *IEEE Micro.*

Prince, C. (2000). Progressive meshes for large models of arbitrary topology. Master's thesis, University of Washington.

Rossignac, J. & Borrel, P. (1993). Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag.

Rossignac, J., Safonova, A., & Szymczak, A. (2001). 3d compression made simple: Edgebreaker on a corner-table. *Shape Modeling International Conference*.

Rubin, S., Bodik, R., & Chilimbi, T. (2002). An efficient profile-analysis framework for data-layout optimizations. *Principles of Programming Languages*.

Ruemmler, C. & Wilkes, J. (1994). An introduction to disk drive modeling. *IEEE Computer*.

Rusinkiewicz, S. & Levoy, M. (2000). Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, pages 343–352.

Sagan, H. (1994). *Space-Filling Curves*. Springer-Verlag.

Schaufler, G., Dorsey, J., Decoret, X., & Sillion, F. (2000). Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, pages 229–238.

Schroeder, W., Zarge, J., & Lorensen, W. (1992). Decimation of triangle meshes. In *Proc. of ACM SIGGRAPH*, pages 65–70.

Scott, N., Olsen, D., & Gannett, E. (1998). An overview of the visualize fx graphics accelerator hardware. *The Hewlett-Packard Journal*, pages 28–34.

Sen, S., Chatterjee, S., & Dumir, N. (2002). Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49:828–858.

Shaffer, E. & Garland, M. (2001). Effient adaptive simplification of massive meshes. In *Proc. of IEEE Visualization*.

Shaffer, E. & Garland, M. (2005). A multiresolution representation for massive meshes.

Sillion, F. (1994). Clustering and volume scattering for hierarchical radiosity calculations. In *Fifth Eurographics Workshop on Rendering*, pages 105–117, Darmstadt, Germany.

Silva, C., Chiang, Y.-J., Correa, W., El-Sana, J., & Lindstrom, P. (2002). Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

Tan, T.-S., Chong, K.-F., & Low, K.-L. (1999). Computing bounding volume hierarchies using model simplification. In *ACM Symposium on Interactive 3D Graphics*, pages 63–70.

Teller, S. J. (1992). *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley.

van Emde Boas, P. (1977). Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*

van Kreveld, M., van Oostrum, R., Bajaj, C., Pascucci, V., & Schikore, D. R. (1997). Contour trees and small seed sets for isosurface traversal. In *Symp. on Computational Geometry*.

Velho, L. & de Miranda Gomes, J. (1991). Digital halftoning with space filling curves. In *ACM SIGGRAPH*, pages 81–90.

Vitter, J. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, pages 209–271.

Wald, I., Dietrich, A., & Slusallek, P. (2004). An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering*.

Weghorst, H., Hooper, G., & Greenberg, D. (1984). Improved computational methods for ray tracing. *ACM Transactions on Graphics*, pages 52–69.

Weiler, K. (1985). Edge-based data structures for solid modeling in a curved surface environment. *IEEE Comput. Graph. Appl.*, 5(1):21–40.

Williams, N., Luebke, D., Cohen, J. D., Kelley, M., & Schubert, B. (2003). Perceptually guided simplification of lit, textured meshes. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

Wilson, A., Larsen, E., Manocha, D., & Lin, M. C. (1999). Partitioning and handling massive models for interactive collision detection. *Computer Graphics Forum (Proc. of Eurographics)*, 18(3):319–329.

Wonka, P., Wimmer, M., & Schmalstieg, D. (2000). Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82.

Wonka, P., Wimmer, M., & Sillion, F. (2001). Instant visibility. In *Proc. of Eurographics*.

Woo, M., Neider, J., & Davis, T. (1997). *OpenGL Programming Guide, Second Edition*. Addison Wesley.

Xia, J., El-Sana, J., & Varshney, A. (1997). Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183.

Yoon, S., Salomon, B., Lin, M. C., & Manocha, D. (2004a). Fast collision detection between massive models using dynamic simplification. *Eurographics Symposium on Geometry Processing*, pages 136–146.

Yoon, S., Salomon, B., & Manocha, D. (2003). Interactive view-dependent rendering with conservative occlusion culling in complex environments. *Proc. of IEEE Visualization*.

Yoon, S.-E., Lindstrom, P., Pascucci, V., & Manocha, D. (2005a). Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH*.

Yoon, S.-E., Salomon, B., Gayle, R., & Manocha, D. (2004b). Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, pages 131–138.

Yoon, S.-E., Salomon, B., Gayle, R., & Manocha, D. (2005b). Quick-VDR: Out-of-core View-dependent Rendering of Gigantic Models. *IEEE Trans. on Visualization and Computer Graphics*.

Zhang, H., Manocha, D., Hudson, T., & Hoff, K. (1997). Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*.